Université de Nancy I
U.E.R. de Mathématiques

Sc.N.75/127 A

L'ANALYSE SYNTAXIQUE DANS FACE

LANGAGE POUR L'ECRITURE DES COMPILATEURS

UNIVERSITÉ NANCY BIBLIOTHÈQUE

Thèse soutenue le 15 mars 1975

par

Hubert PISTRÉ

pour l'obtention du Doctorat de Spécialité de
Mathématiques Appliquées (Informatique)

JURY :

M.	C.	PAIR	Président
Mme	F.	BELLEGARDE	Examinateurs
M.	J.C.	DERNIAME	
Mme	C.	ROLLAND	

Université de Nancy I
U.E.R. de Mathématiques

L'ANALYSE SYNTAXIQUE DANS FACE

LANGAGE POUR L'ECRITURE DES COMPILATEURS

Thèse soutenue le 15 mars 1975

par

Hubert PISTRÉ

pour l'obtention du Doctorat de Spécialité de
Mathématiques Appliquées (Informatique)

JURY :

M.	C.	PAIR	Président
Mme	F.	BELLEGARDE	Examinateurs
M.	J.C.	DERNIAME	
Mme	C.	ROLLAND	

"VIVRE, C'EST S'ENGAGER A AGIR,
VIVRE, C'EST ETRE SOI,
C'EST RESISTER ET AIMER,
ACCEPTER ET REFUSER,
VIVRE, C'EST CREER."

(Martin Gray)

A ma femme
qui m'a supporté, encouragé et aidé
tout au long de ce travail.

Monsieur le Professeur C. PAIR, Directeur de l'Institut Universitaire de Calcul Automatique, est à l'origine du projet FACE ; il a inspiré et dirigé le présent travail et me fait l'honneur de présider ce jury. Je lui adresse aujourd'hui toute ma reconnaissance pour sa constante bienveillance, la clarté de son enseignement et l'apprentissage que je lui dois dans les travaux de recherche.

Je remercie Madame C. ROLLAND pour l'honneur qu'elle me fait en acceptant d'être membre de ce jury.

Que Monsieur J.C. DERNIAME trouve ici l'expression de mes vifs remerciements pour l'attrayante initiation à l'informatique et la formation qu'il m'a données, ainsi que pour l'intérêt qu'il me manifeste en critiquant ce travail.

J'adresse mon amicale reconnaissance à Madame F. BELLEGARDE, pour son excellente collaboration, la qualité de nos discussions et sa présence dans ce jury.

De même, mes remerciements vont à Monsieur J. MAROLDT et à Mademoiselle A.M. BOUCHET pour leurs travaux au sein du projet FACE.

Je remercie Madame J. CHABRIER, Messieurs M. DORION et J.M. PIERREL pour leur contribution à l'implémentation de l'analyse descendante, ainsi que Monsieur F. DEBERDT, Mademoiselle L. GALLIER, Madame D. MARCHAND et l'équipe du secrétariat de l'I.U.T. Informatique pour la diligence et la qualité de leur travail ayant permis la réalisation de cet ouvrage.

S O M M A I R E

CHAPITRE 1. Place de l'analyse syntaxique dans FACE

1.1.	Le processus de compilation en FACE	I.1
1.2.	L'analyseur dans le programme FACE	I.6
1.3.	Principe du traitement des grammaires	I.6
1.4.	Interface	I.8

CHAPITRE 2. Solutions envisagées

2.1.	Principe général	II.1
2.2.	Présentation des notions utilisées dans la suite	II.1
2.3.	L'analyseur descendant	II.4
2.4.	Présentation de l'algorithme d'analyse descendante	II.12
2.5.	L'analyse ascendante	II.14
2.6.	Les données de l'analyseur ascendant	II.17
2.7.	L'algorithme d'analyse ascendante	II.19
2.8.	Autres analyseurs	II.21
2.9.	Les symboles terminaux en FACE	II.23

CHAPITRE 3. Les données du préanalyseur

3.1.	Description générale	III.1
3.2.	Le vecteur de renseignements	III.2
3.3.	La table des méta-symboles : TMS	III.3
3.4.	La table des terminaux : TDT	III.3

3.5. Le graphe représentant la grammaire : GRG III.5

3.6. Exemple de représentation d'une grammaire III.7

3.7. La table des caractères à sauter : TCS III.9

3.8. La table morphologique III.10

CHAPITRE 4. Etude de la grammaire par FADA
Adéquation d'une analyse descendante

4.1. Organisation générale du préanalyseur IV.1

4.2. Outils utilisés par FADA IV.3

4.3. Gestion de la mémoire par FADA IV.10

4.4. Suppression de la récursivité à gauche directe : 1PASSAGE IV.11

4.5. Recherche des règles vides : VIDFINI IV.14

4.6. Construction des relations $\mathcal{T}$, $\mathcal{Y}$ et $\mathcal{F}$: 2PASSAGE IV.19

4.7. Construction des fermetures transitives $\mathcal{T}^*$ et $\mathcal{F}^*$: WARSHALL IV.26

4.8. Calcul des relations de précédence de la relation vs : VOISEVAL IV.27

4.9. Contrôle de la grammaire en vue d'une analyse descendante : VERIF IV.30

4.10. Contrôle des terminaux : sous-programme EMPILER IV.36

4.11. Contrôle des caractères à sauter : TESTSAU IV.41

4.12. Le sous-programme : FINAL IV.42

4.13. Le sous-programme : ECRIRE IV.42

CHAPITRE 5. Le sous-programme d'analyse descendante ANALDES

5.1. Les données de l'analyseur V.1

5.2. Le module d'initialisations MODINIT1 V.4

5.3. Principe de l'algorithme d'analyse V.6

5.4. Algorithme d'analyse V.7

5.5. Le sous-programme LIRE (c en fonction de NDI) V.10

5.6. Le sous-programme SASAUTER V.11

5.7. Le sous-programme ERREURD V.11

CHAPITRE 6. Etude de la grammaire Adéquation d'une analyse ascendante

6.1. Principaux éléments d'une grammaire destinée à l'analyse ascendante VI.1

6.2. Format général d'une grammaire en format ascendant VI.8

6.3. Organisation générale de l'étude en vue de l'analyse ascendante VI.9

6.4. Etude des conflits de précédence : le sous-programme PRECEDEN VI.12

6.5. Construction de la nouvelle représentation de la grammaire : le sous-programme RETOURN VI.13

6.6. Réorganisation de la mémoire : le sous-programme REORG VI.22

6.7. Contrôle de la grammaire : le sous-programme VERIFA VI.23

CHAPITRE 7. L'analyseur ascendant ANALAS

7.1. Les données de l'analyseur VII.1

7.2. Principe de l'algorithme d'analyse VII.1

7.3. Le module d'initialisations MODINIT2 VII.2

7.4. L'algorithme d'analyse ascendante VII.4

7.5. Le module LIRA (c, B, r1, r2) VII.7

7.6. Les modules ERREUR et ERREURR VII.8

CHAPITRE 8. Analyseur descendant et ascendant

8.1. Algorithme d'analyse ascendante - descendante, récursif et indéterministe VIII.3

8.2. Exemple d'utilisation de l'algorithme VIII.8

PROLONGEMENTS POSSIBLES.

ANNEXE 1. Procédures : CALL A.1.1

SAVE A.1.3

RETURN A.1.7

RESTAUR A.1.10

ANNEXE 2. Les sous-programmes :

LECT(NDI) A.2.1

MESSERR(NO) A.2.14

AVANCARR A.2.21

ANNEXE 3. Le sous-programme :

REØRG A.3.1

ANNEXE 4. Le sous-programme :

CØNSEC A.4.1

ANNEXE 5. Deuxième partie du sous-programme :

VERIFA A.5.1

CHAPITRE I

PLACE DE L'ANALYSE SYNTAXIQUE DANS FACE

1.1. LE PROCESSUS DE COMPILATION EN FACE.

Ce travail est une contribution à l'implémentation du langage FACE, langage permettant l'écriture de compilateurs [1].

Comme la plupart des autres langages permettant l'écriture de compilateurs, FACE s'applique à des langages dont la syntaxe est définie par une grammaire à contexte libre. Un programme écrit en FACE est essentiellement constitué de la description d'une (ou plusieurs) grammaires G complétées d'une description de la traduction du langage sous la forme de procédures attachées aux règles des grammaires.

L'analyse syntaxique permet la transformation du texte source en une ramification dont les noeuds sont étiquetés par des numéros de règles de la grammaire G. Cette ramification permet d'induire un ordre d'exécution des différentes procédures attachées aux règles de grammaire, permettant la compilation du texte source [2].

oOo

L'un des objectifs du langage FACE est de libérer le plus complètement possible l'utilisateur du problème de l'analyse syntaxique et même de la méthode d'analyse. Cet aspect de FACE est l'objet du présent travail.

L'analyse syntaxique est effectuée automatiquement par une procédure standard d'analyse, faisant partie du système, ayant pour données, une grammaire et le texte source à analyser. De plus le programmeur peut ignorer la méthode d'analyse. Aussi l'objectif ultime est-il de réaliser le choix automatique de la méthode d'analyse en fonction de la grammaire utilisée.

Le résultat de l'analyse et sa représentation doivent donc être indépendants de la méthode d'analyse adoptée.

Le résultat de l'analyse syntaxique est une ramification. Sa représentation doit permettre une utilisation aisée et efficace ; cette dernière consiste en l'appel successif des procédures de compilation. La représentation postfixée sous forme de n^{os} de règles répond à la question [2] et a été adoptée.

oOo

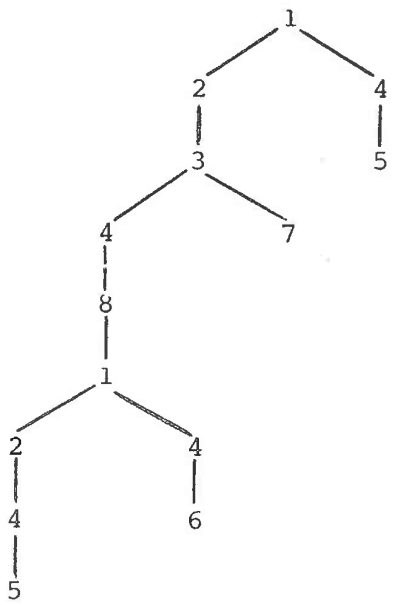
A partir d'un texte source et d'une grammaire du langage, (grammaire à contexte libre), il est possible d'obtenir par analyse syntaxique une ramification engendrée par la grammaire qui admette le texte source comme mot des feuilles. On convient d'étiqueter les noeuds de la ramification par les numéros de règles.

Exemple :

Soit la grammaire d'axiome E

E ::= E + T / 1 / T / 2
T ::= T * F / 3 / F / 4
F ::= a / 5 / b / 6 / c / 7 / (E) / 8

Les numéros figurant après / sont les numéros de règle.
La phrase (a + b) * c + a appartient au langage engendré et son analyse syntaxique fournit la ramification :



Cette ramification peut se décrire par composition des sous-ramifications qu'elle contient :

$r = 1 \times (2 \times 3\ (4 \times 8 \times 1 \times (2 \times 4 \times 5 + 4 \times 6) + 7) + 4 \times 5)$

(Pour une étude formelle des ramifications on pourra se reporter à [12] et [13]).

La compilation du texte source peut se décrire par une procédure qui admet la ramification r en paramètre : comp(r). Cette procédure effectue des actions (génération du texte objet, construction et consultation de tables). Elle doit être décrite récursivement car $comp(A \times (r_1 + r_2 \ldots + r_q))$ dépend de $comp(r_1)$, $comp(r_2), \ldots, comp(r_q)$. La procédure comp fournit un résultat qui peut être utilisé dans l'appel précédent de comp :

$comp(A \times (r_1 + r_2 \ldots + r_q)) = F(comp(r_1), comp(r_2), \ldots, comp(r_q))$.

F est une procédure qui utilise les résultats de $comp(r_1)$, $comp(r_2)$... et donne le résultat de l'appel en cours de comp. Elle est liée à la règle de grammaire qui a engendré cette partie de la ramification. Ainsi dans l'exemple

$$comp(r) = F_1(comp(r_1), comp(r_2))$$

F_1 est liée à la règle 1 de la grammaire.

$comp(r_1)$ et $comp(r_2)$ ont pour résultat l'adresse et le type du résultat des sous-expressions (a + b) × c et a. F_1 effectue les contrôles de type et génère des instructions de conversion (éventuellement) et l'addition des deux résultats qu'elle a en paramètre.

La compilation de l'expression (a + b) × c + a s'écrit donc :

$F_1(F_2(F_3(F_4(F_8(F_1(F_2(F_4(F_5))),F_4(F_5))),F_7)),F_4(F_5))$.

Les procédures F_5, F_6, F_7 attachées aux règles conduisant aux terminaux n'ont pas de paramètres. Leur rôle est simplement de fournir en résultat l'adresse et le type des variables a, b, etc.

Les procédures F_i doivent être récursives, ceci présentant une difficulté de réalisation pratique. Mais il est facile de supprimer la récursivité en transmettant les résultats sur une pile. Introduisons les procédures G_i :

G_i = empiler (F_i (sommet$_i$)) ;

la procédure sommet$_i$ prélève un certain nombre de résultats au sommet de la pile (ce nombre dépend de i) et les fournit en paramètres à F_i. Les résultats de F_i sont ensuite empilés.

La compilation de l'expression précédente s'écrit alors :

$G_5;G_4;G_2;G_6;G_4;G_1;G_8;G_4;G_7;G_3;G_2;G_5;G_4;G_1$.

Pour générer un programme objet en SYMBØL-10070, ces procédures s'écrivent :

G_5 = empiler l'adresse de la variable a
G_6 = empiler l'adresse de la variable b
G_7 = empiler l'adresse de la variable c
$G_2 = G_4 = G_8$ = procédures vides
G_3 = (x_1=sommet de la pile ; x_2=sommet de la pile ;

```
si x1<16 alors début r:=x1 ; générer(MW,x1 x2) ;
                      si x2<16 alors i:=i-1
                fin
          sinon début si x2<16 alors début r:=x2 ;
                                          générer(MW,x2 x1)
                                    fin
                                sinon début r:=i:=i+1 ;
                                          générer(LW,i x1) ;
                                          générer(AW,i x1)
                                    fin
                fin ;
    empiler (r))
```

G_1 est identique à G_3 en remplaçant MW par AW.

(Pour simplifier, les 3 variables a, b, c sont supposées être de même type et les registres en nombre suffisant).

La variable globale i doit être initialisée à -1 au début.

Ainsi la compilation s'effectue au moyen de procédures attachées aux règles de la grammaire. Ces procédures communiquent entre elles résultats et paramètres au moyen d'une pile. L'ordre de l'appel de ces différentes procédures se déduit de la ramification admettant le texte source comme mot des feuilles. Plus exactement l'ordre de l'appel des procédures est celui d'une représentation postfixée de la ramification.

Pour plus de précision sur ce principe on pourra se reporter à [2].

Le langage FACE met en oeuvre des concepts qui permettent d'utiliser facilement ce principe.

Il permet d'une part d'écrire des grammaires à contexte libre en affectant un numéro aux règles. D'autre part il permet de déclarer des rangées de procédures. Ces procédures peuvent prendre des paramètres et ranger des résultats sur une pile qui est gérée automatiquement. Le programmeur dispose en outre des 3 procédures standards :

- ANALYSE recevant en paramètre une grammaire, cette procédure effectue l'analyse du texte source et produit une ramification représentée en notation postfixée par une rangée d'entiers.

- APPL reçoit en paramètre une rangée d'entiers et une rangée de procédures. Pour chaque entier de la rangée d'entiers APPL appelle la procédure de la rangée de rang correspondant à l'entier. APPL se charge donc d'appeler dans le bon ordre les procédures G_i en fonction du résultat d'une analyse.

- ANAPPLYSE combine les effets de ANALYSE et de APPL en appelant les procédures de la rangée au fur et à mesure que l'analyse produit des n^{os} de règle.

Le principe de compilation qu'offre FACE a déjà été utilisé pour la réalisation du compilateur FACE. Ce compilateur a en effet été écrit en FACEO, sous-ensemble de FACE, assez simple pour que son compilateur ait pu être écrit en Métasymbol.

1.2. L'ANALYSEUR DANS LE PROGRAMME FACE.

Cette brève présentation montre bien que l'algorithme d'analyse syntaxique en FACE est complètement indépendant de l'utilisation du résultat de l'analyse.

Rien ne s'oppose donc à ce que le sous-programme d'analyse syntaxique dispose de plusieurs algorithmes et tente de choisir le meilleur algorithme pour chaque cas particulier.

1.3. PRINCIPE DU TRAITEMENT DES GRAMMAIRES.

Les grammaires écrites par l'utilisateur font l'objet d'une compilation avant d'être utilisées par les procédures d'analyse syntaxique. Cette compilation a 3 objectifs :

- s'assurer que la description des grammaires est correcte et qu'elles peuvent faire l'objet d'une analyse au moyen des algorithmes dont on dispose. Si ces conditions ne sont pas vérifiées, des messages seront émis de manière à permettre des corrections.

- choix d'un algorithme d'analyse syntaxique adapté à chaque grammaire.

- construire une représentation des grammaires permettant une analyse aussi efficace que possible. Le choix de l'algorithme influe d'ailleurs sur la représentation adoptée. Cette représentation des grammaires fera partie du programme objet généré puisqu'elle sera exploitée par les procédures standards d'analyse à chaque exécution du programme FACE.

Le compilateur FACE contrôle la validité syntaxique de la description des grammaires et transforme chacune d'elles en une

rangée d'éléments, la représentant. Cette transformation, comme le reste du compilateur FACE, est écrite en FACEO [1] et [3]. Cette représentation est ensuite étudiée et transformée par un sous-programme standard FADA (Fabrication Automatique des Données de l'Analyseur), appelé pour chaque grammaire.

FADA s'assure que la grammaire permet des analyses syntaxiques, compte tenu des algorithmes dont on dispose (émission éventuelle de diagnostics), choisit une méthode d'analyse et transforme la représentation en une représentation adaptée à cette méthode. Enfin, cette représentation est placée sur un support secondaire à accès direct afin de libérer l'espace mémoire.

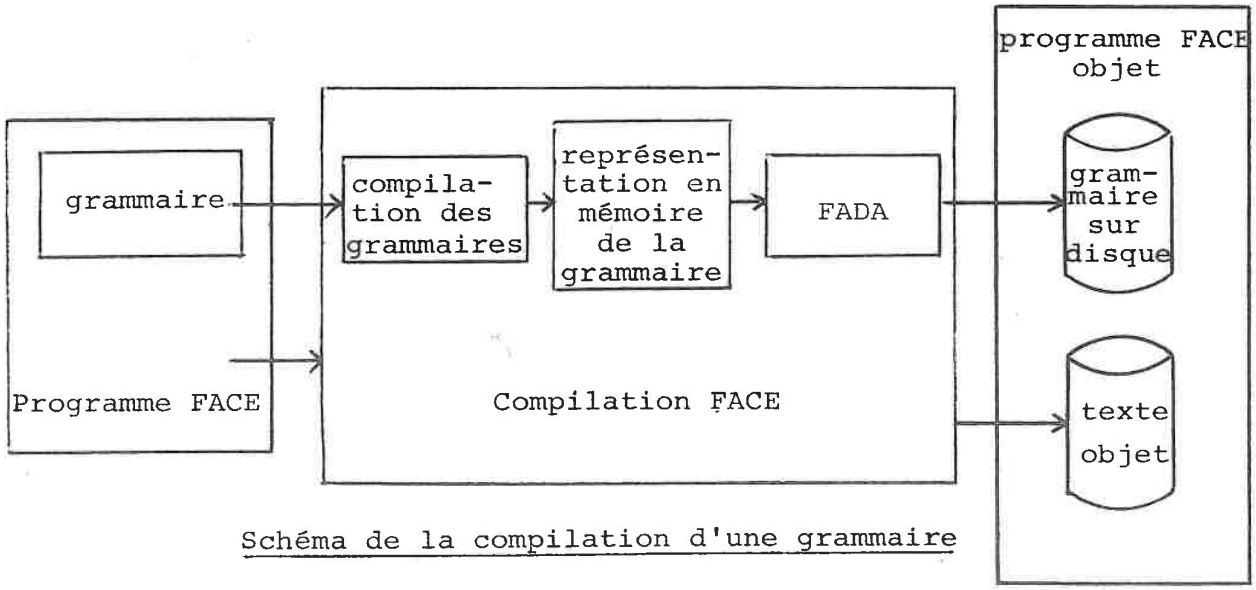


Schéma de la compilation d'une grammaire

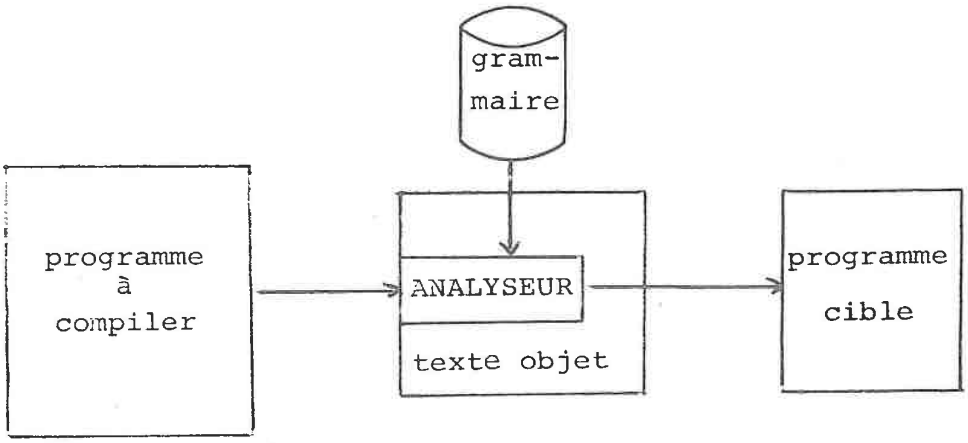


Schéma de l'exécution d'un programme FACE

Bien entendu, la"compilation des grammaires" est une partie du compilateur FACE écrit en FACEO. Ce compilateur utilise donc la procédure d'analyse syntaxique avec la grammaire FACE en entrée. Cette grammaire n'est pas écrite dans le compilateur FACE car FACEO ne supporte pas les grammaires.

La représentation en mémoire de la grammaire FACE a donc été écrite à la main, sous forme d'un programme Métasymbol [4] en utilisant uniquement les instructions de réservation de constantes de l'assembleur. Le sous-programme FADA lui a été appliqué afin d'obtenir une représentation sur disque de la grammaire FACE, utilisée à chaque compilation d'un programme FACE.

Les procédures standards FADA, ANALYSE, ANAPPLYSE sont écrites en Métasymbol.

Remarque : Dans toute la suite, sauf indication spéciale, on désignera indistinctement sous les termes : "analyseur" ou "procédure d'analyse" les procédures standards ANALYSE et ANAPPLYSE Ces procédures sont en effet physiquement représentées par le même programme. Elles ne diffèrent que par l'utilisation qu'elles font des résultats produits par l'algorithme d'analyse.

1.4. INTERFACE.

a) Interface entre la procédure FADA et le compilateur FACE.

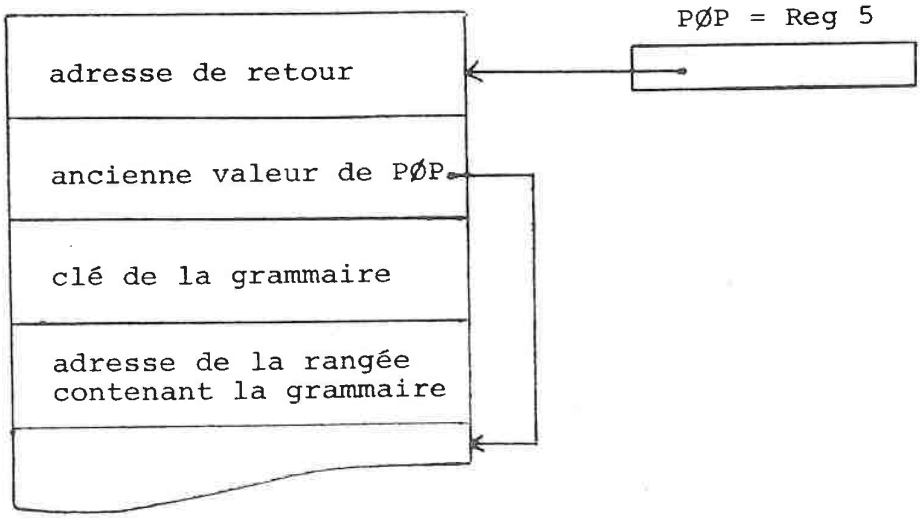
Le compilateur FACE appelle la procédure FADA en lui transmettant deux paramètres :

- une rangée d'éléments contenant le résultat de la compilation d'une grammaire ;

- un numéro ou clé caractéristique de la grammaire traitée. Le compilateur FACE attribue un numéro propre à chaque grammaire figurant dans le programme FACE. La procédure FADA place les grammaires sur un fichier disque. Toutes les grammaires d'un même programme FACE sont placées dans un même fichier parti-

tionné. Chacune d'elles constitue un membre du fichier ayant pour nom de membre la clé fournie par le compilateur FACE.

Les paramètres sont transmis à FADA au sommet de la pile temporaire gérée pendant l'exécution d'un programme FACEO. L'adresse du sommet de la pile se trouve dans le registre 5(PØP). A l'appel de FADA la structure du sommet de la pile est la suivante :



Le registre PØP n'est pas modifié par FADA. La pile est gérée automatiquement par le programme FACEO.

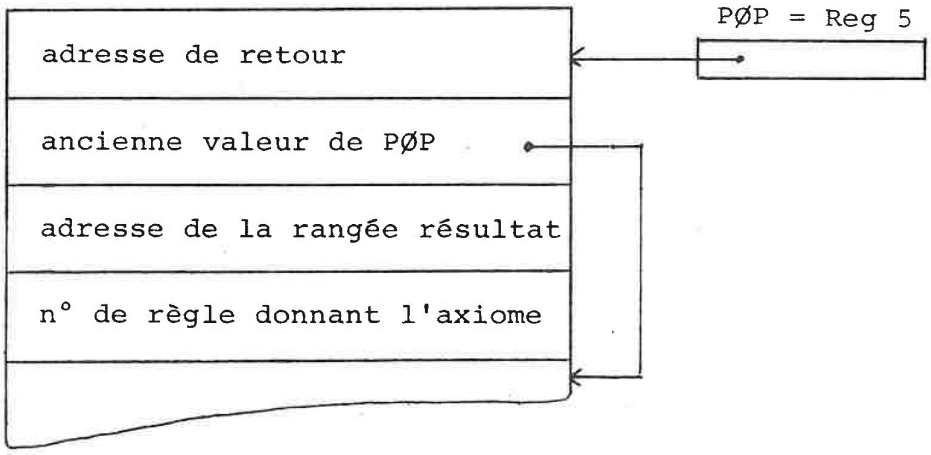
b) Interface entre la procédure ANALYSE et le programme FACE.

Le programme FACE appelle la procédure ANALYSE en lui transmettant deux paramètres :

- la grammaire avec laquelle le texte source doit être analysé et son axiome. Ces deux renseignements sont fournis par le seul numéro de règle dont le membre de gauche est l'axiome. Ce numéro est suffisant pour déterminer la grammaire contenant cette règle, car, dans un programme FACE, on doit utiliser des ensembles disjoints de n° de règles dans les différentes grammaires.

- la rangée d'entiers devant recevoir le résultat de l'analyse (représentation postfixée de la ramification).

Ces paramètres sont transmis de la même manière que ceux de FADA, au sommet de la pile gérée pendant l'exécution d'un programme FACE :



La procédure ANALYSE effectue la correspondance entre le n° de règle donnant l'axiome et la grammaire contenant cette règle grâce à la zone des grammaires générée par le compilateur FACE. La zone des grammaires est une table définie à l'adresse TG (définition externe) qui associe pour chaque grammaire son numéro de règle le plus élevé à la clé attribuée par le compilateur FACE à cette grammaire. Ces deux renseignements figurent dans deux mots consécutifs. Le premier mot de la zone des grammaires est sa taille en mots.

Par un examen séquentiel de la zone des grammaires jusqu'à un numéro de règle supérieur ou égal à celui qui lui est donné en paramètre, ANALYSE obtient la clé de la grammaire qu'il doit charger en mémoire.

Exemple : TG

6	150	1	200	2	310	3

Il y a 3 grammaires dans le programme source. Si ANALYSE reçoit en paramètre le numéro de règle 252, il utilisera la grammaire de clé 3.

c) Interface entre la procédure ANAPPLYSE et le programme FACE.

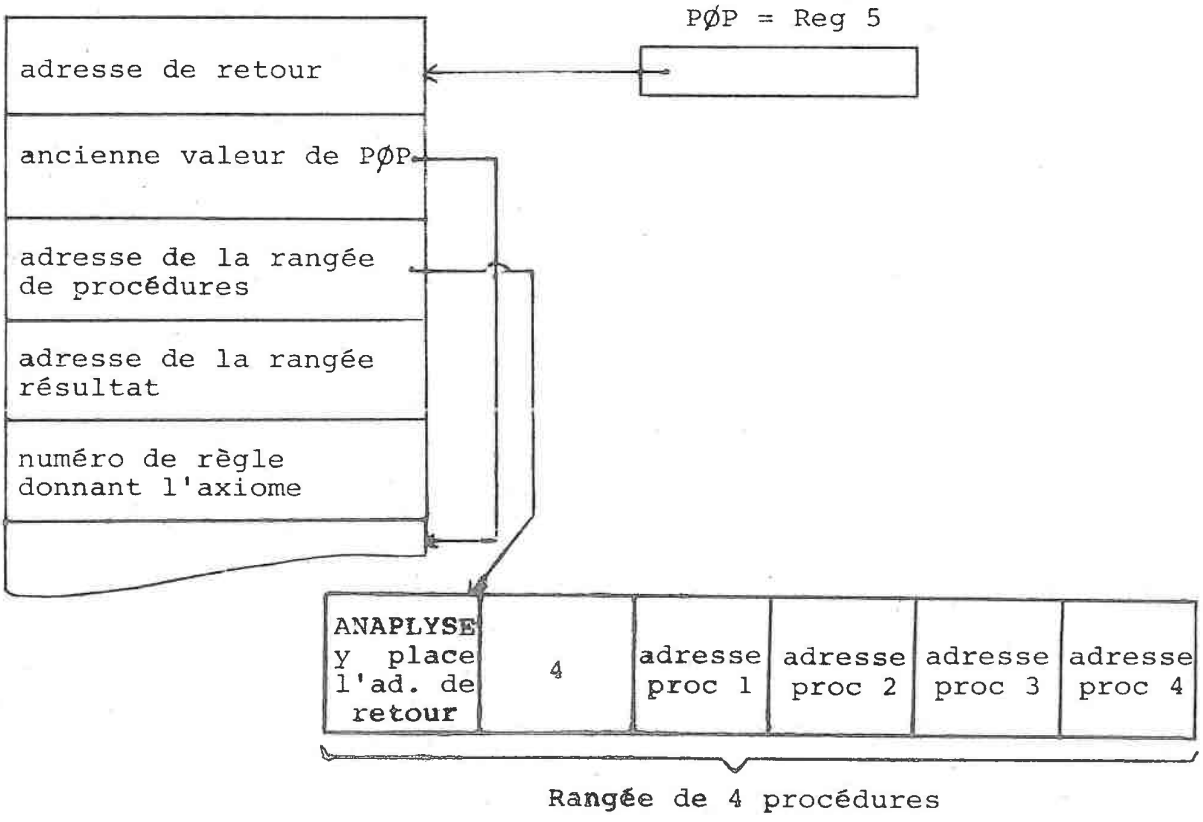
La procédure ANAPPLYSE a trois paramètres :

- le numéro de règle donnant accès à la grammaire et à l'axiome (comme ANALYSE).

- la rangée d'entiers devant recevoir le résultat de l'analyse (comme ANALYSE).

- la rangée de procédures à activer au fur et à mesure de la génération de la chaîne postfixée. ANAPPLYSE trouve sur la pile, l'adresse d'une table donnant les adresses de début d'exécution de chaque procédure (un mot par procédure de la rangée). Le premier mot de cette table est destiné à recevoir l'adresse de retour de chaque procédure dans ANAPPLYSE. Le second mot contient le nombre de procédures de la rangée. Le troisième mot donne l'adresse de début de la première procédure, etc...

A l'appel de ANAPPLYSE, l'état de la pile est donc :



C H A P I T R E II

SOLUTIONS ENVISAGEES

2.1. PRINCIPE GENERAL.

L'un des principaux objectifs de l'analyseur est d'être aussi général que possible, de manière que l'utilisateur n'ait pas (ou pas trop) à se préoccuper des problèmes d'analyse syntaxique lorsqu'il écrit une grammaire.

Un deuxième objectif est l'efficacité. Il est important que les programmes écrits en FACE soient efficaces. L'analyseur étant appelé une ou plusieurs fois au cours de l'exécution, ses performances sont importantes.

En conséquence, il a semblé préférable de s'orienter vers la réalisation de plusieurs algorithmes pouvant chacun s'adapter à un type particulier de grammaire, plutôt que vers la réalisation d'un analyseur universel, pouvant admettre une gamme très large de grammaires mais forcément plus lourd.

Les deux exigences ci-dessus sont bien sûr contradictoires, et à moins de disposer d'une gamme très étendue d'algorithmes, la première exigence ne peut être entièrement satisfaite et ne l'est pas actuellement, dans la mesure où un nombre suffisant d'algorithmes n'est pas implémenté.

Les algorithmes envisagés jusqu'alors sont tous déterministes (pas de retour en arrière ou d'analyses parallèles) afin de satisfaire à la 2ème exigence.

Cette manière d'aborder le problème présente en outre l'avantage de permettre une conception modulaire. Il est toujours envisageable d'étudier l'intégration de nouveaux algorithmes au système.

2.2. PRESENTATION DES NOTIONS UTILISEES DANS LA SUITE.

Une grammaire à contexte libre G est un quadruplet (T,N,::=,X) où T est le vocabulaire terminal (ensemble des symboles terminaux ou terminaux) ;

N le vocabulaire non terminal (ensemble des symboles non terminaux ou non terminaux) ;

T et N sont deux ensembles finis disjoints ;

On notera V l'ensemble $T \cup N$;

$X \in N$ est l'axiome de la grammaire G.

::= est une relation binaire appelée relation de production entre N et V^* (ensemble des mots sur V).

Un couple (A,α) tel que $A::=\alpha$ s'appelle une règle. Le nombre de règles est fini.

Les notions de ramification et de langage engendrés par une grammaire G, déjà utilisées au chapitre I, sont supposées connues.

Il est possible de définir entre les éléments de V et de V^* un certain nombre de relations qui seront utiles dans la suite.

a) Relations de dérivation.

La relation "se réécrit" notée $\succ\!\!\!-$ est définie dans V^*:

$\alpha \succ\!\!\!- \beta \Longleftrightarrow (\exists A \in N,\ \exists \lambda \in V^*,\ \exists \lambda' \in V^*,\ \exists \mu \in V^*)(\alpha = \lambda A \lambda',\ \beta = \lambda \mu \lambda' \text{ et } A::=\mu)$

La fermeture transitive de $\succ\!\!\!-$ est notée $\overset{*}{\succ\!\!\!-}$.
La fermeture transitive stricte de $\succ\!\!\!-$ est notée $\overset{+}{\succ\!\!\!-}$.
Lorsque $\alpha \overset{*}{\succ\!\!\!-} \beta$ on dit que β dérive de α ou que de α dérive β.
Lorsque $\alpha \overset{+}{\succ\!\!\!-} \beta$ on dit que β dérive strictement de α.

Le langage engendré par une grammaire G est l'ensemble des mots de T^* qui dérivent de l'axiome.

b) La relation initiale.

Soit $\mathcal{J}$ la relation dans V définie par :

$a \mathcal{J} B \Longleftrightarrow (\exists \varphi \in V^*,\ \exists \psi \in V^*)\ (A::=\varphi B \psi \text{ et } \varphi \overset{*}{\succ\!\!\!-} \Lambda)$;

Λ représente la chaîne vide.

La fermeture transitive de la relation $\mathcal{I}$ est notée $\mathcal{I}^*$.
La fermeture transitive stricte de la relation $\mathcal{I}$ est notée $\mathcal{I}^+$.
Lorsque $A\,\mathcal{I}^*\,B$ on dit que B est initiale de A.
Lorsque $A\,\mathcal{I}^+\,B$ on dit que B est initiale stricte de A.
S'il existe un symbole qui est initiale stricte de lui-même, la grammaire est récursive à gauche.

Dans la suite, on notera $\mathcal{I}(A)$ l'ensemble des symboles $B\in V$ tels que $A\mathcal{I}B$
$\mathcal{I}^*(A)$ l'ensemble des symboles $B\in V$ tels que $A\mathcal{I}^*B$
$\mathcal{I}^+(A)$ l'ensemble des symboles $B\in V$ tels que $A\mathcal{I}^+B$

$A\,\mathcal{I}^*\,B \Longleftrightarrow A \overset{*}{\longrightarrow} B\ \varphi \quad \varphi \in V^*$

c) La relation finale.

Soit la relation dans V définie par :

$A\,\mathcal{F}\,B \Longleftrightarrow (\exists\ \varphi\in V^*,\ \exists\ \psi\in V^*)\ (A::=\varphi\ B\ \psi \text{ et } \psi \overset{*}{\longrightarrow} \Lambda)$

Comme pour la relation $\mathcal{I}$ il est possible de définir la fermeture transitive $\mathcal{F}^*$ de $\mathcal{F}$ et sa fermeture transitive stricte $\mathcal{F}^+$, ainsi que les ensembles $\mathcal{F}(A)$, $\mathcal{F}^*(A)$, $\mathcal{F}^+(A)$.

$A\mathcal{F}^+\,B \Longleftrightarrow A \overset{*}{\longrightarrow} \varphi\ B \quad \varphi \in V^*.$

d) La relation successeur.

Soit $\mathcal{S}$ la relation dans V définie par :

$A\,\mathcal{S}\,B \Longleftrightarrow (\exists\ C\in N,\ \exists\ \varphi\in V^*,\ \exists\ \mu\in V^*,\ \exists\ \psi\in V^*)\ (C::=\varphi A\mu B\psi \text{ et } \mu \overset{*}{\longrightarrow} \Lambda$

e) La relation de voisinage.

Soit vs la relation dans V définie par $vs = \mathcal{F}^{-1*}\,\mathcal{S}\,\mathcal{I}^*$
$\mathcal{F}^{-1}$ est la relation inverse de $\mathcal{F}$; $\mathcal{F}^{-1*}$ en est la fermeture transitive.

$A\ vs\ B \Longleftrightarrow (\exists\ C\in V,\ \exists\ D\in V)\ (C\,\mathcal{S}\,D \text{ et } C\,\mathcal{F}^*\,A \text{ et } D\,\mathcal{I}^*\,B)$
$A\ vs\ B \text{ et } A\in T \text{ et } B\in T \Longleftrightarrow$ les terminaux A et B peuvent se suivre dans un mot du langage

2.3. L'ANALYSEUR DESCENDANT.

Le premier algorithme étudié fut un analyseur descendant déterministe. Il accepte les grammaires de type LL (1) avec factorisation, mais autorise en plus la récursivité à gauche directe.

Comme on va le voir dans le chapitre suivant, cet algorithme qui a été le premier envisagé, a influencé la définition de l'interface entre le préanalyseur FADA et le reste du compilateur FACE. Ainsi la représentation des grammaires dont il a été question en 1.2 est bien adaptée à une analyse descendante. Cette représentation sera transformée par le préanalyseur s'il décide d'adopter un autre algorithme.

L'algorithme de base sur lequel est fondé cet analyseur est décrit dans le cours de C. PAIR page 72 [2]. Le choix entre les différentes règles de réécriture possibles se fait en fonction d'un caractère lu à l'avance.

La grammaire est représentée par une ramification.

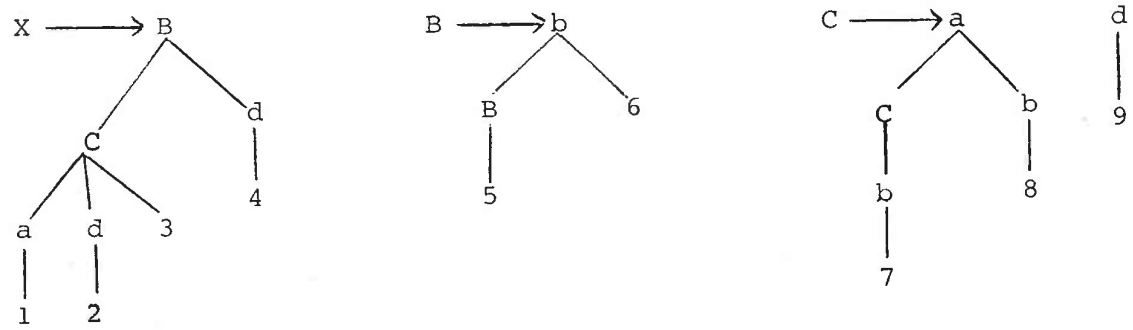
Pour tout non terminal A on a besoin de représenter les mots λk tels que $A::=\lambda$ soit la règle de n° k. On utilise pour cela une ramification sur $V \cup R$ ($V = N \cup T$ ensemble des symboles terminaux et non terminaux de la grammaire ; R ensemble des n^{os} de règles), tels que les mots λk soient les chemins joignant une racine à une feuille.

Exemple :

Soit la grammaire :

X::=BCa/1/BCd/2/BC/3/Bd/4/
B::=bB/5/b/6/
C::=aCb/7/ab/8/d/9/

La ramification correspondante est :



Ce type de représentation partage la représentation des débuts de règles identiques. Cela permet de n'avoir à choisir entre deux règles qu'au dernier moment.

Ainsi dans cet exemple, l'analyse de X commence systématiquement par analyse (B). L'analyse de B étant faite, il faut choisir entre analyse (C) et analyse (d) et ainsi de suite.

Dans cette perspective, les conditions que doit vérifier la grammaire sont les suivantes :

- La grammaire doit être réduite inférieurement ($\forall A \in N, \exists \alpha \in T^*, A \overset{*}{\rightarrow} \alpha$).

- Pour que l'algorithme soit borné, la grammaire ne doit contenir aucun symbole récursif à gauche.

- Pour que l'algorithme puisse être déterministe, il faut que, lorsqu'un choix entre plusieurs règles se présente, le choix puisse être fait en fonction du 1er caractère à analyser qui se présente dans le texte source (caractère lu à l'avance).

Il faut distinguer entre deux sortes de choix :

1) Choix entre les règles : $A ::= \alpha B \beta$ et $A ::= \alpha C \gamma$.

($A \in N \quad B, C \in N \cup T \quad \alpha, \beta, \gamma \in (N \cup T)^*$)

Le choix se fera seulement lorsque α est analysé. Le terminal c lu à l'avance au moment du choix est donc une initiale de B ou de C. Il permettra donc de choisir à coup sûr entre B et C, si et seulement si les symboles B et C n'ont aucune initiale terminale commune, soit $\mathcal{J}^*(B) \cap \mathcal{J}^*(C) = \emptyset$.

(Remarque : aucune initiale terminale commune $\Rightarrow$ aucune initiale commune)

2) Choix entre les règles : $A::=\alpha B\beta$ et $A::=\alpha$

Une fois α analysé, il faut choisir entre analyse(B) et fin de règle. On ne peut choisir analyse(B) que si $B\ \mathcal{J}^*\ c$. On ne peut choisir la fin de règle que si c peut suivre une locution dérivée de A, c'est-à-dire $A\ \mathcal{F}^{-1*}\ \mathcal{F}\ \mathcal{J}^*\ c$ soit A vs c. Le choix sera donc déterminé si $\mathcal{J}^*(B) \cap vs(A) = \emptyset$.

D'où les deux règles de déterminisme que doit vérifier la grammaire :

$\forall A \in N \quad \forall \alpha, \beta, \gamma \in (N \cup T)^* \quad \forall B, C \in N \cup T$

(D1) $A::= \alpha B \beta \ / \ \alpha C \gamma \Longrightarrow \mathcal{J}^*(B) \cap \mathcal{J}^*(C) = \emptyset$

(D2) $A::= \alpha B \beta \ / \ \alpha \Longrightarrow \mathcal{J}^*(B) \cap vs(A) = \emptyset$

En outre ces deux règles assurent que l'algorithme est borné car une grammaire vérifiant ces règles ne peut être récursive à gauche. En effet, supposons que dans une telle grammaire, il existe un non terminal A récursif à gauche. Alors $A\ \mathcal{J}^+\ A$.

$(\exists A_0, A_1 \ldots A_n \in N)\ (A_0\ \mathcal{J}\ A_1 \ldots \mathcal{J} A_n\ \mathcal{J}\ A)$ avec $A_0 = A$.

La grammaire étant réduite inférieurement $(\exists \alpha \in T^*)$ $(A \overset{*}{\succ} \alpha)$, soit a le premier terminal du mot α :

$A\ \mathcal{J}^*\ a \qquad a \in T$

$\exists B_0, B_1 \ldots B_m \in V$ tels que :

$B_0\ \mathcal{J}\ B_1 \ldots \mathcal{J} B_m$ avec $B_0 = A$ et $B_m = a$.

$(A_O, A_1 \ldots A_n)$ et $(B_O, B_1 \ldots B_m)$ sont deux chemins du graphe $(V,\mathcal{J})$. Ces deux chemins ont une origine commune (A) mais leurs extrémités sont différentes. L'extrémité $B_m = a \in T$ du 2ème chemin ne peut appartenir au premier.

Soit B_ℓ le premier point du deuxième chemin n'appartenant pas au premier. $\exists\, i \in [O,n]$ tel que $A_i = B_{\ell-1}$

$A_i ::= A_{i+1}\ \alpha$ (si $i = n$ $A_{i+1} = A$)

$A_i ::= B_\ell\ \beta$

$A_{i+1} \mathcal{J}^*\ A^*$ et $B_\ell\ \mathcal{J}^*\ a$.

Or $A\ \mathcal{J}^*$ a donc $a \in \mathcal{J}^*(A_{i+1}) \cap \mathcal{J}^*(B_\ell)$ ce qui est contradictoire avec la règle de déterminisme (D1).

Cas particulier des règles vides.

Dans ce qui précéde il a été implicitement supposé qu'il n'y avait pas de règle vide dans la grammaire (grammaire sans Λ).

Cependant il est très important d'admettre la possibilité d'écrire des règles vides dans une grammaire. En effet, en FACE, les actions de compilation réalisées par les procédures d'une rangée sont provoquées par la sortie d'un numéro de règle et ne peuvent donc avoir lieu qu'en fin de règle. Dans d'autres langages d'écriture de compilateurs (META 5 par exemple [15]) il est possible d'écrire des appels de sous-programmes (des actions) au milieu des règles, ce qui est souvent intéressant en compilation. On dispose des mêmes possibilités en FACE grâce aux règles vides.

Lorsqu'on écrit en META 5 :

```
A = B .action 1  C  .action 2
```

On écrira en FACE :

```
A : B X C / 2
X : / 1
```

La procédure de rang 1 réalise l'action 1, celle de rang 2 l'action 2.

Pour tenir compte des règles vides dans l'analyse syntaxique, on note les non terminaux d'où peut dériver la chaîne vide.

Au moment de l'analyse, il est possible que le terminal c lu à l'avance ne soit initiale d'aucune des alternatives qui se présentent. Lorsqu'il se présente une alternative d'où dérive la chaîne vide, il est possible de choisir en regardant les initiales de leurs successeurs. Il est alors demandé à l'analyseur un travail interprétatif supplémentaire. Le travail est évité si on n'a jamais à choisir entre plusieurs non terminaux d'où dérive la chaîne vide ou entre un non terminal d'où dérive la chaîne vide et une fin de règle.

Les conditions que doit satisfaire la grammaire sont alors les suivantes :

$\forall A \in N,\ \forall \alpha,\beta,\gamma \in (N \cup T)^*$ $\quad \forall B,C \in N \cup T$

(D'1) $A ::= \alpha\ B\ \beta\ /\ \alpha\ C\ \gamma \Rightarrow (\mathcal{T}^*(B) \cap \mathcal{T}^*(C) = \emptyset)$
et non $(B \overset{*}{\rightarrowtail} \Lambda$ et $C \overset{*}{\rightarrowtail} \Lambda)$

(D'2) $A ::= \alpha\ B\ \beta\ /\ \alpha \Rightarrow \mathcal{T}^*(B) \cap vs(A) = \emptyset$
et non $B \overset{*}{\rightarrowtail} \Lambda$.

Dans ce cas, si le terminal c lu à l'avance n'est initiale d'aucune des alternatives qui se présentent, on choisit une alternative d'où dérive la chaîne vide s'il en existe.

Pour faciliter alors le travail de l'analyseur, il convient lorsqu'une suite d'alternatives comporte un non terminal d'où dérive la chaîne vide, de placer ce non terminal en dernière alternative.

C'est dans ce cas que nous nous sommes placés. Dans le cas où il existe des règles

$A ::= \alpha\ B\ \beta\ /\ \alpha\ C\ \gamma$ avec $B \overset{*}{\rightarrowtail} \Lambda$ et $C \overset{*}{\rightarrowtail} \Lambda$

ou

$A ::= \alpha\ B\ \beta\ /\ \alpha$ avec $B \overset{*}{\rightarrowtail} \Lambda$

il est possible de transformer la grammaire en remplaçant la règle A ::= α B β par

A ::= α B β / α β

ces deux règles portant le même numéro.

Alors le symbole B n'est analysé que lorsqu'il conduit à une chaîne non vide.

Il est possible d'introduire dans la représentation de la règle supplémentaire introduite entre les chaînes α et β, des symboles fictifs portant les numéros des règles permettant d'analyser la chaîne vide à partir de B. Ces symboles fictifs sont du même type que ceux introduits ci-après pour tenir compte de la récursivité à gauche directe.

La récursivité à gauche.

La récursivité à gauche est interdite par les règles de déterminisme. Il a cependant semblé intéressant de permettre la récursivité à gauche directe, c'est-à-dire des règles du genre :

$A ::= A\ \alpha_1 / A\ \alpha_2 \ldots\ldots / A\ \alpha_m / \beta_1 / \beta_2 \ldots\ldots / \beta_n$

avec

$A \in N \qquad \alpha_1, \alpha_2 \ldots\ldots \alpha_n\ \beta_1 \ldots\ldots \beta_n \in (N \cup T)^*$

$\beta_1 \ldots \beta_n$ n'admettant pas A comme facteur gauche.

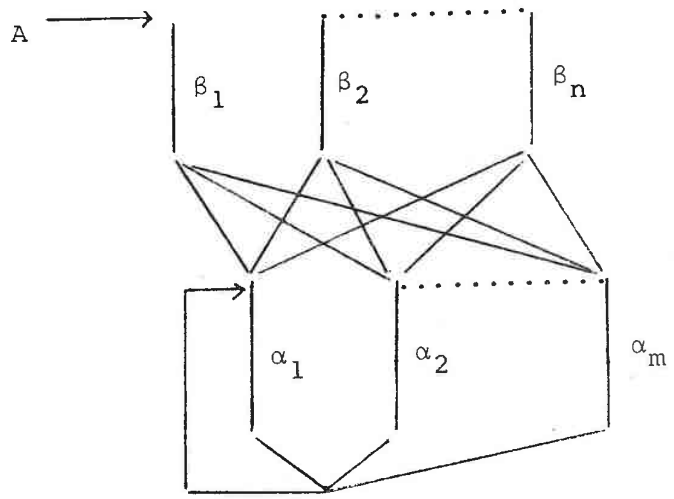
Ce type de règle est a priori interdit par la règle de déterminisme (D1).

Le principe de la solution consiste à transformer ces règles en

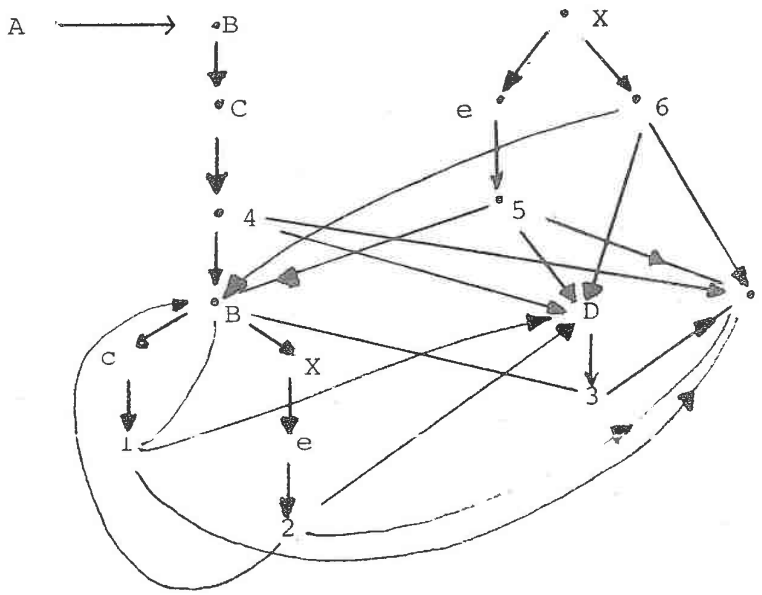
$A ::= (\beta_1 / \beta_2 \ldots\ldots / \beta_n)\ (\alpha_1 / \alpha_2 \ldots\ldots / \alpha_m)^*$

La représentation des règles de réécriture de A n'est

alors plus une arborescence mais un graphe que l'on peut schématiser par :



Exemple : A::=ABc/1/ABXe/2/AD/3/BC/4/Xe/5/X/6/



fin de règle sans numéro

On constate que l'on introduit sur ce chemin un nouveau type de point : le numéro de règle avec successeur. Il correspond à la sortie d'un numéro de règle mais avec poursuite de la règle.

La présence de ces numéros de règles avec successeur permet de générer la même représentation postfixée que celle que l'on obtient à l'aide des règles initiales récursives à gauche. En effet, un symbole A récursif à gauche (récursivité directe) a pour règles de réécriture :

$$A ::= A\ \alpha_1 / i_1 \ \ldots / A\ \alpha_m / i_m / \beta_1 / j_1 / \ldots / \beta_n / j_n$$

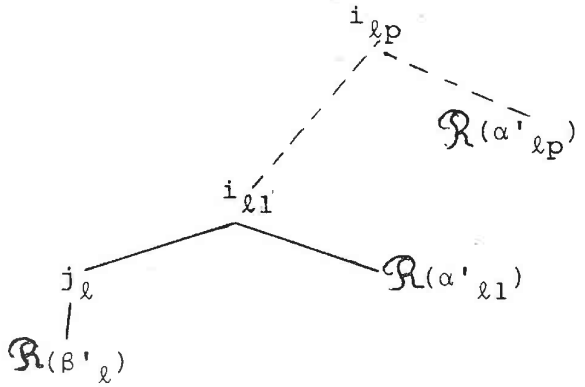
$i_1 \ldots i_m$, $j_1 \ldots j_n$ étant les numéros de règles.

Toute phrase $\varphi \in T^*$ dérivée de A a la forme :

$$\varphi = \beta'_\ell\ \alpha'_{\ell 1} \ldots \alpha'_{\ell p}$$

avec $p \geqslant 0 \quad \ell \in [0,n] \quad (\forall i \in [1,p])\ (\ell_i \in [0,m])$

$$\beta_\ell \overset{*}{\rightarrowtail} \beta'_\ell \qquad \alpha_{\ell i} \overset{*}{\rightarrowtail} \alpha'_{\ell i}$$



$\mathcal{R}(\alpha')$ étant la ramification engendrée par la grammaire résultant de la dérivation de α en α'

La représentation postfixée de cette ramification est :

$$p(\mathcal{R}(\beta'_\ell))\ j_\ell\ p(\mathcal{R}(\alpha'_{\ell 1}))\ i_{\ell 1} \ldots p(\mathcal{R}(\alpha'_{\ell p}))\ i_{\ell p}$$

$p(\mathcal{R}(\alpha'))$ est la représentation postfixée de la ramification $\mathcal{R}(\alpha')$.

Si le même mot y est analysé par la grammaire transformée (à savoir : $A ::= (\beta_1 / j_1 / \ldots / \beta_n / j_n /)\ (\alpha_1 / i_1 \ldots \alpha_m / j_m)^*$) l'analyse de β'_ℓ produit $p(\mathcal{R}(\beta'_\ell))$ puis le numéro j_ℓ est sorti par rencontre du point "numéro de règle avec successeur".

L'analyse de $\alpha'_{\ell 1}$ produit alors $p(\mathcal{R}(\alpha'_{\ell 1}))$. Le numéro de règle avec successeur $i_{\ell 1}$ est alors rencontré et sorti et ainsi de suite... On obtient donc bien la même représentation postfixée.

Les règles de grammaire récursives transformées et représentées de cette manière doivent vérifier les règles de déterminisme (D1) et (D2). On doit donc avoir :

$$J^*(\beta_1) \cap J^*(\beta_2) \ldots \cap J^*(\beta_n) = \emptyset$$

et

$$J^*(\alpha_1) \cap J^*(\alpha_2) \ldots \cap J^*(\alpha_m) \cap vs(A) = \emptyset.$$

Dans l'exemple cela se traduit par :

$$J^*(B) \cap J^*(X) = \emptyset$$

$$e \cap J^*(B) \cap J^*(D) \cap vs(A) = \emptyset$$

$$c \cap J^*(X) = \emptyset.$$

2.4. PRESENTATION DE L'ALGORITHME D'ANALYSE DESCENDANTE.

Dès que l'on est assuré que la grammaire vérifie les règles D1 et D2, tout choix peut se faire à l'aide de la relation J^* .

En effet, si un choix entre un symbole et une fin de règle du type $A ::= \alpha\ B\ \beta\ /\ \alpha$ se présente, lorsque α est analysé, si $B\ J^*\ c$, on choisit la première règle, sinon on choisit la 2ème règle.

En fait il se peut que $c \notin vs(A) \cup J^*(B)$, mais alors c'est une erreur qui sera détectée ultérieurement.

L'algorithme d'analyse a donc besoin de la relation J^* (J^* est une partie de $N \times (T \cup \Lambda)$), et d'une représentation de la grammaire. La ramification de la grammaire est en fait représentée par une ramification binaire. Chaque noeud qui n'est pas une feuille est l'origine de deux accès : le lien "suc" (successeur dans la règle) le lien "alt" (alternative).

Dans une suite d'alternatives on convient de mettre en fin :

- les fins de règles ;
- les non terminaux d'où peut dériver la chaîne vide.

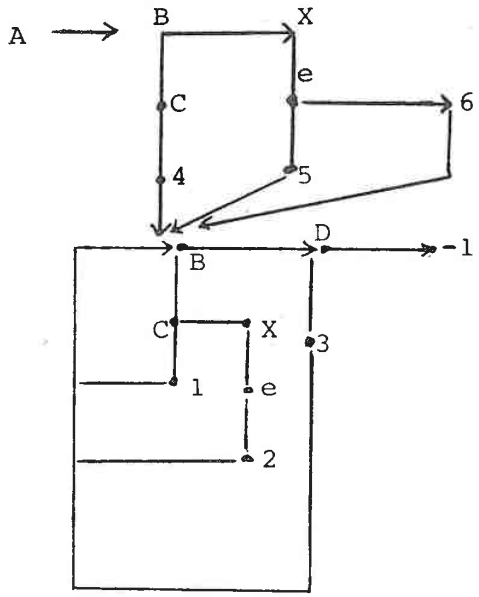
Exemples de représentations :

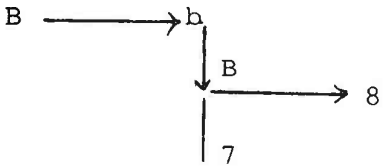
Les règles :

A ::= A B c / 1 / A B X e / 2 / A D / 3 / B C / 4 / X e / 5 / X / 6 /

B ::= b B / 7 / b / 8 /

seront représentées par





sont représentés :

verticalement le lien "suc"

horizontalement le lien "alt"

Ce graphe comporte 4 types de noeuds :

- les noeuds étiquetés par des non terminaux
- les noeuds étiquetés par des terminaux
- les feuilles portant des numéros de règle
- les noeuds portant des numéros de règle et admettant un successeur.

Les numéros de règle sont des nombres positifs ou nuls. Le numéro -1 indique une fin de règle ne portant pas de numéro.

L'algorithme d'analyse s'écrit alors (en notant accès la fonction donnant accès à partir d'un non terminal à ses règles de réécriture) :

```
début
procédure analyse (A) ;
début B:=accès(A) ;
re:si B est un non terminal alors début si B⇒*c ou B⇒*Λ alors
                                        début analyse(B) ;
                                              B:=suc(B) ;
                                              allera re
                                        fin
                                  fin
                            sinon si B est un terminal alors
                                   début si B=c alors
                                          début lire(c) ;
                                                B:=suc(B) ;
                                                allera re
                                          fin
                                   fin
                                  sinon début co B est un n° de règle co
                                              si B≠-1 alors sortir(B) ;
                                              si B admet un successeur
                                                                  alors
                                               début B:=suc(B) ;
                                                     allera re
                                               fin ;
                                              allera fin
                                        fin ;
   si B admet une alternative alors début B:=alt(B) ;
                                        allera re
                                   fin ;
    erreur ;
fin:fin co fin de la procédure analyse co
    lire(c) ; analyse(axiome) ; si le mot à analyser n'est pas
                                entièrement lu alors erreur
  fin
```

2.5. L'ANALYSE ASCENDANTE.

L'algorithme de base sur lequel est fondé cet analyseur est un algorithme à contexte borné (1,1), décrit dans le cours de C. PAIR [2. page 101]. Le choix entre les différentes règles

possibles se fait en fonction d'un terminal lu à l'avance et du symbole qui se trouve au sommet de la pile d'analyse. L'algorithme consiste à construire dans la pile d'analyse des locutions par lecture et empilement des terminaux du mot à analyser et réduction de la partie droite de la pile.

L'algorithme est borné si et seulement si
- il n'y a pas de règle vide
- aucun symbole ne peut dériver de lui-même.

Il est cependant possible d'admettre des grammaires contenant des règles vides (voir II.16).

L'algorithme général contient deux sortes de choix :
1) A chaque itération choix entre une lecture (empilement du caractère lu à l'avance et lecture d'un nouveau caractère) et une réduction transformant un facteur droit de la pile d'analyse en un symbole non terminal.
2) Lorsque l'on décide d'effectuer une réduction, choix d'une règle.

Ces choix doivent être rendus déterministes.

Soit B le symbole se trouvant au sommet de la pile et c le terminal lu. Si l'on effectue une lecture on devra avoir $B \mathcal{F} J^* c$, soit $B \lessdot c$. Si l'on effectue une réduction on devra avoir $B \mathcal{F}^{-1+} \mathcal{F} J^* c$, soit $B \gtrdot c$. Le choix pourra donc être déterministe si ces deux relations ne sont pas vérifiées en même temps. D'où la première condition de déterminisme :

(D3) $\forall B \in N \cup T \quad \forall c \in T \quad$ non $B \lessdot c$ et $B \gtrdot c$

Si l'on effectue une réduction, il faut déterminer le facteur droit de la pile (λ) qui va être réduit, puis choisir la règle $A ::= \lambda$.

(D4) s'il existe deux règles $A ::= \lambda$ et $A_1 ::= \lambda' B \lambda$ on doit avoir :
non$(B \lessdot A)$ ou $vs(A) \cap vs(A_1) = \emptyset$

(D5) s'il existe deux règles $A_1::=\lambda$ et $A_2::=\lambda$
on doit avoir :

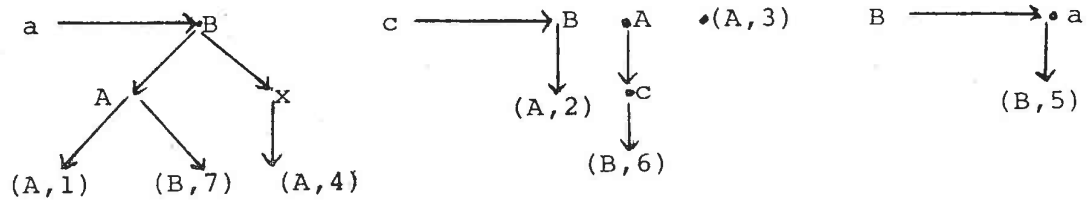
$vs(A_1) \cap vs(A_2) = \emptyset$ ou $\nexists B \in N \cup T$ tel que $B \prec A_1$ et $B \prec A_2$

Représentation de la grammaire.

Afin de pouvoir effectuer les réductions à partir de la connaissance du sommet de la pile, il est nécessaire d'avoir accès aux règles à partir du dernier symbole du membre de droite. La représentation de la grammaire sera toujours une ramification, mais "inverse" (en quelque sorte) de celle adoptée pour l'analyse descendante.

Exemple : soit la grammaire :
A::=ABa/1/Bc/2/c/3/xBa/4/

B::=aB/5/cAc/6/ABa/7/



Les feuilles donnent le membre de gauche et le n° de règle.

Cas des règles vides.

L'existence de règles vides pose un problème, car il est possible à tout moment d'effectuer une réduction de la chaîne vide consistant en l'introduction au sommet de la pile d'un non terminal A tel que A::=$\hat{A}$. Cela peut conduire à un calcul infini.

Pratiquement une telle réduction n'est possible que si : $B \prec A$ et A vs c (B sommet de la pile et c=caractère lu).

Il faut donc effectuer une réduction dans deux cas :

ou bien $B \gtrdot c$

ou bien $\exists$ A, A::=Λ et A vs c.

On est donc conduit à étendre la relation $\gtrdot$

$B \gtrdot c \Longleftrightarrow B\mathcal{F}^{-1+}\mathcal{G}\mathcal{J}^{*}c$ ou $\exists A \in N$ A::=Λ $B \lessdot A$ et A vs c.

Lorsque, au cours de l'analyse on décide d'effectuer une réduction, on essaie normalement de réduire un facteur droit non vide de la pile par accès aux règles qui se terminent par l'élément se trouvant au sommet de la pile. Dans le cas où la chaîne vide peut être réduite, une information supplémentaire accompagnera la relation $\gtrdot$ dans la matrice de précédence, signifiant que l'on doit chercher la réduction à faire parmi les règles vides de la grammaire.

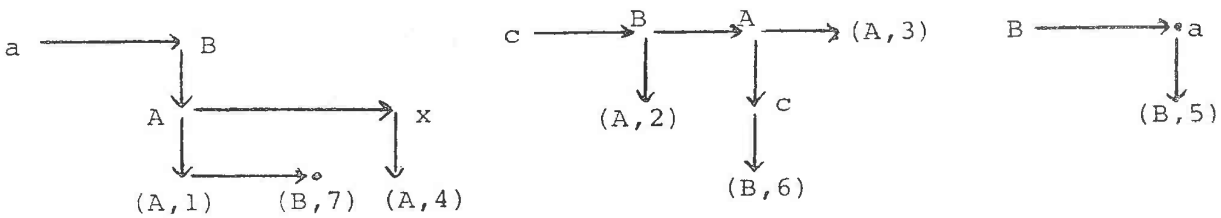
Dans la mesure où la grammaire contient plusieurs règles vides, lorsqu'il est décidé de réduire la chaîne vide, il faut choisir l'une de ces règles. Ces règles ont toutes le même membre de droite. Le choix pourra donc être déterministe si elles respectent la règle (D5).

2.6. LES DONNEES DE L'ANALYSEUR ASCENDANT.

Les données de l'analyseur ascendant comportent une représentation de la grammaire sous forme d'une ramification.

Cette ramification est en fait représentée par une ramification binaire avec lien "suc" et lien "alt".

Les ramifications de l'exemple précédent deviennent :



Lorsque des règles vides existent dans la grammaire, on y accède par un symbole spécial Λ. En ajoutant à la grammaire précédente les règles A::=Λ/8/ et B::=Λ/9/, la ramification ci-dessus est complétée par

$$\Lambda \longrightarrow (A,8) \longrightarrow (B,9).$$

En plus de la grammaire l'analyseur utilise les relations de précédence $\lessdot$ et $\gtrdot$. Ces relations sont données sous forme de matrices. Afin de simplifier le programme et de diminuer l'espace mémoire nécessaire, ces relations ne sont données qu'entre des symboles quelconques et des terminaux (les deux relations sont des parties (disjointes) de $(N \cup T) \times T$ et non de $(N \cup T) \times (N \cup T)$). Cela suffit pour effectuer le choix entre lecture, réduction, réduction de la chaîne vide. Lorsqu'une réduction est à effectuer cela suffit dans de nombreux cas pour choisir la réduction (chaque fois que le caractère c suffit à la détermination de la règle). Cela est insuffisant dans la mesure où il existe des règles du type :

$(A_1 ::= \lambda,\ A_2 ::= \lambda 1 \lambda,\ \lambda 1 \in (N \cup T)^*,\ \lambda \in (N \cup T)^+)$

ou $(A_1 ::= \Lambda\ A_2 ::= \Lambda)$ et $\exists\, c \in vs(A_1) \cap vs(A_2)$.

Dans la mesure où l'on doit choisir entre A_1 et A_2 et où le caractère lu appartient à $vs(A_1) \cap vs(A_2)$ le choix ne peut se faire qu'en fonction de la relation de précédence liant le nouveau sommet de pile et A_1 ou A_2. Ce fait est signalé par l'existence de la relation $*$ entre A_1 et c, A_2 et c.

On peut alors trouver dans une table spéciale les symboles B tels que $B \lessdot A_1$ ou $B \lessdot A_2$. L'accès à cette table se fait à partir des symboles A_1 et A_2. Elle n'est construite que pour les non terminaux A tels qu'il existe $c \in T$ $A * c$.

Lorsque l'analyse commence, la pile est vide. Pour amorcer l'algorithme on introduit un symbole spécial (noté $\longrightarrow\!\!\dashv$) dans la pile. Ce symbole vérifie les relations :

$$\longrightarrow\!\!\dashv \lessdot A \Longleftrightarrow X \rceil^* A$$

X étant l'axiome de la grammaire.

On considère que la phrase à analyser se termine par une marque de fin (notée ⊢—— ; pratiquement ce caractère est généré par la rencontre de la fin de fichier) ; on a

$X \lessdot \vdash\!\!\!-\!\!\!-$

$B \gtrdot \vdash\!\!\!-\!\!\!- \Longleftrightarrow X \mathcal{F}^{+} B.$

Ces relations de précédence avec les marqueurs de début et de fin ne peuvent pas être fournies à l'analyseur car en FACE, l'axiome n'est connu qu'au moment de l'analyse, à l'exécution du programme FACE. L'analyseur devra donc dans une phase d'initialisation calculer ces relations.

2.7. L'ALGORITHME D'ANALYSE ASCENDANTE.

```
début calcul des relations de précédence concernant les marqueurs
      de début et de fin ;
   h:=1 ; B:=PILE[1]:= ⊢—— co initialisation de la pile par le
                                marqueur de début co ;
   lec:lire(c) ;
   iter:si pas de relation entre B et c alors ERREUR
        si B ≺ c alors co lecture co
                    début si c= ⊢—— alors
                                début si h≠2 alors ERREUR
                                           sinon allera fin
                                fin ;
                          h:=h+1 ; B:=PILE[h]:=c ; allera lec
                    fin ;
        si B ≻ c alors co réduction co
                    début D:=accès(B) ; h:=h-1 ; B:=PILE[h] ;
                    essai:si D est une feuille de la ramification alors
```

```
                début si D ≺ c ou D ≻ c alors
                        début si D * c alors
                                début si non B ≺ D alors
                                        allera alt
                                fin ;
                                h:=h+1 ; B:=PILE[h] :=D ;
                                allera iter
                        fin ;
                alt:si D n'a pas de lien "alt"
                        alors ERREUR ;
                        D:=alt(D) ; allera essai
        fin ;
    si B = D alors début h:=h-1 ; B:=PILE[h] ;
                        D:=suc(D) ;
                        allera essai
                fin ;
    allera alt
fin ;
co on a la relation B ≻ c ; réduction de la chaîne vide co
D:=accès(Λ) ; allera essai
fin : fin
```

On constate facilement que les conditions imposées aux grammaires par cet algorithme sont plus faibles que celles imposées aux grammaires de simple précédence [5,6] ou de précédence faible [7,8].

2.8. AUTRES ANALYSEURS.

Après l'étude complète des deux analyseurs présentés en 2.4 et 2.7 on trouvera au chapitre 8 l'ébauche d'un analyseur acceptant des grammaires dont chaque règle vérifie soit les conditions (D'1) et (D'2) soit les conditions (D3), (D4) et (D5). Il s'agit encore d'un analyseur rendu déterministe par lecture d'un caractère à l'avance.

L'ensemble de ces 3 analyseurs permet de traiter une large gamme de grammaires LR(1) [9,10] tout en mettant en oeuvre des algorithmes assez simples et efficaces. Il n'est bien entendu pas exclu d'adjoindre des analyseurs plus complexes (éventuellement avec retour arrière ou calculs parallèles). D'un autre côté l'utilisateur peut aussi modifier sa grammaire de manière à ce qu'elle puisse être traitée par les analyseurs existants.

Dans la mesure où des analyseurs plus complexes seraient disponibles, le préanalyseur FADA pourrait signaler à l'utilisateur les conflits exigeant l'utilisation d'un analyseur complexe. De cette façon, l'utilisateur aurait un moyen d'améliorer l'efficacité du compilateur qu'il réalise.

o o
o

Une autre optique pourrait être de tenter de modifier automatiquement la grammaire écrite par l'utilisateur, de manière qu'elle rentre dans les cadres envisagés. C'est ce qui est fait pour la récursivité à gauche directe dans l'analyse descendante.

Il faut cependant que la chaîne postfixée fournie par l'analyseur soit bien celle correspondant à la grammaire qu'il reçoit.

Cependant certaines modifications de la grammaire, dans le cadre de l'algorithme d'analyse ascendante, peuvent permettre d'accroître sensiblement la gamme de grammaires acceptées.

Fisher a démontré [11] que l'élimination des conflits de précédence simple est toujours possible pour toute grammaire à contexte libre. Mais la transformation d'une grammaire permettant

l'élimination des conflits de précédence simple est difficile à réaliser. Elle est beaucoup plus simple pour les relations de précédence adoptées ici(précédence faible).

Supposons que les symboles A et B présentent un conflit de précédence : $A \lessdot B$ et $A \gtrdot B$. On introduit une règle supplémentaire A'::=A permettant de conserver la relation $A \gtrdot B$ et d'éliminer la relation $A \lessdot B$.

Les règles du type $U::=\alpha A \varphi C \beta$ avec $\varphi \overset{*}{\gtrdot} A$ et $C \overset{*}{\gtrdot} B\,\psi$

sont remplacées par : $U::=\alpha A' \varphi C \beta$.

Cela revient à introduire une réduction supplémentaire du symbole A en A'. La règle A'::=A ne devra bien sûr pas porter de n° de règle de manière à ne pas modifier la chaîne postfixée produite par l'analyse.

Il est facile de démontrer que cet algorithme supprime tous les conflits de précédence et n'en introduit aucun autre. En effet :

1) Il ne peut plus y avoir de conflit entre A et B puisque toutes les relations $A \lessdot B$ ont été supprimées.

2) Il ne peut y avoir conflit entre A' et un symbole X quelconque. En effet on ne peut avoir que $A' \lessdot X$. Pour que la relation $A' \gtrdot X$ existe il faudrait que A' figure à la fin d'une règle, ce qui n'est pas le cas.

3) S'il existe des conflits entre un symbole X et A' ($X \lessdot A'$ et $X \gtrdot A'$), il ne s'agit pas de nouveaux conflits mais de conflits qui existaient entre X et A (et peuvent être supprimés par l'introduction de X'::=X).

On dispose donc d'un moyen permettant d'éliminer les conflits de précédence. Cependant il a l'inconvénient d'introduire des parties droites de règles communes.

Supposons en effet qu'un conflit $A \lessdot B$ et $A \gtrdot B$ soit résolu par l'introduction de la règle A'::=A.

Puisque l'on avait la relation A $\gtrdot$ B, il existait dans la grammaire une règle se terminant par A : X::=φA et X vs B.

Lorsque A se trouve au sommet de la pile, il faut savoir le réduire, c'est-à-dire choisir entre les règles A'::=A et X::=φA. Ce choix ne peut pas être effectué en fonction du terminal lu à l'avance. En effet :

$$B \in vs(X) \cap vs(A')$$

Si $\varphi = \varphi_1 X_1$ on devra donc avoir non($X_1 \lessdot A'$), ce qui revient à dire que dans les règles de type (1), α ne doit pas se terminer par X_1, ou si $\alpha = \Lambda$ non($X_1 \lessdot U$).

Si $\varphi = \Lambda$ on ne devra pas avoir Y $\lessdot$ X et Y $\lessdot$ A' soit α ne doit pas se terminer par Y ou si $\alpha = \Lambda$ non(Y $\lessdot$ U).

2.9. LES SYMBOLES TERMINAUX EN FACE.

Les symboles terminaux des grammaires FACE ne sont pas forcément de simples caractères, mais des chaînes de caractères quelconques, définis dans la grammaire. Cela complique sensiblement la procédure lire(c) de lecture d'un terminal introduite dans les algorithmes présentés aux paragraphes 2.4 et 2.7.

On dispose donc au cours de l'analyse d'une table des symboles terminaux. Le rôle de la procédure lec est de reconnaître dans la suite de caractères du texte source qui se présentent l'un de ces terminaux. Cette reconnaissance doit bien sûr être déterministe. Des conflits sont possibles si un symbole terminal est facteur gauche d'un autre. Cela existe dans le langage FACE lui-même où l'on utilise

- le symbole : dans les instructions conditionnelles ;
- le symbole = dans les relations ;
- le symbole := dans l'affectation.

On ne peut pas en déduire pour autant que le langage contient des ambiguïtés. Le choix de l'un ou l'autre symbole terminal peut en effet être parfaitement déterminé par le contexte.

Deux approches différentes permettent de rendre la lecture d'un terminal déterministe :

1) La procédure lec lit le plus long terminal possible. Soit t_1 et t_2 deux terminaux tels que t_1 soit facteur gauche de t_2. $t_2 = t_1 x$. Le choix proposé est le bon à condition que $\forall t \in T$ tel que x soit facteur gauche de t ou t soit facteur gauche de x, $t \notin vs(t_1)$.

2) Il n'est pas nécessaire que l'acquisition d'un nouveau terminal soit déterministe relativement à l'ensemble des terminaux de la grammaire, mais seulement relativement à l'ensemble des terminaux susceptibles d'être lus à un instant donné.

a) En analyse descendante :

Après lecture d'un nouveau terminal c, on tente l'analyse d'un symbole B tel que $B \mathfrak{J}^* c$. On doit en fait choisir entre un certain nombre de symboles B et l'on prend celui qui vérifie $B \mathfrak{J}^* c$ (qui doit être unique). Si le choix se présente entre les symboles $B_1 \ldots B_n$, il est nécessaire que la lecture puisse se faire sans ambiguïté entre les terminaux c tels que $c \in T \cap \mathfrak{J}^*(B_1) \cup \mathfrak{J}^*(B_2) \ldots \cup \mathfrak{J}^*(B_n)$.

b) En analyse ascendante :

Le symbole lu doit toujours être un voisin du symbole qui se trouve au sommet de la pile d'analyse. Il est donc nécessaire que la lecture puisse se faire sans ambiguïté entre les terminaux voisins d'un même symbole.

Dans les deux cas, on détermine ainsi des sous-ensembles de symboles terminaux à l'intérieur desquels la lecture de chaque élément doit pouvoir être déterministe.

On peut exiger comme condition de déterminisme que dans chaque sous-ensemble aucun symbole terminal ne soit facteur gauche d'un autre. C'est la solution qui a été adoptée.

Il est possible aussi d'appliquer la condition 1) à chaque sous-ensemble, ce qui est moins strict, à savoir :

pour chaque sous-ensemble R

$\forall t_1, t_2 \in R \quad t_2 = t_1 x \Longrightarrow \forall t \in R \quad x$ facteur gauche de t ou t facteur gauche de $x \Longrightarrow \underline{non}(t_1 \text{ vs } t)$.

2.10. L'ANALYSE LEXICOGRAPHIQUE.

Des essais d'utilisation de FACE ont rendu souhaitable l'étude d'une extension de FACE qui concerne l'analyse syntaxique : l'analyse lexicographique, c'est-à-dire la possibilité de reconnaître par un mécanisme simple et souple des entités syntaxiques fondamentales du texte à traduire comme les nombres ou les identificateurs.

Il s'agit toujours d'entités assez simples ne nécessitant pas pour être analysées un algorithme d'analyse syntaxique du type de ceux qui ont déjà été décrits, ou, ce qui revient au même, pouvant être décrites plus simplement qu'avec des grammaires de Chomsky de type 2.

Ces entités seront toujours des éléments d'un certain langage régulier, analysable par un automate fini. Elles sont cependant insérées au sein d'un texte plus complexe qui globalement ne peut être décrit que par une grammaire de Chomsky de type 2.

Ainsi les algorithmes d'analyse syntaxique tout en conservant leur généralité doivent admettre que certaines parties de la phrase à analyser aient une structure plus simple, et avoir recours pour celles-ci à un procédé d'analyse plus efficace et mieux adapté permettant surtout d'éviter des indéterminismes en étendant le contexte à droite.

Une manière assez simple d'obtenir ce résultat, aussi bien dans la définition de FACE que dans les algorithmes d'analyse syntaxique, consiste à élargir la définition des terminaux.

De cette manière l'adjonction de l'analyse lexicographique dans l'analyse syntaxique ne modifie pas la procédure d'analyse proprement dite, mais seulement l'algorithme

d'acquisition d'un terminal (procédure lect dans les algorithmes précédents). L'analyse lexicographique sera donc bien séparée, fonctionnellement de l'analyse syntaxique et pourra s'appliquer sans modification aux différents algorithmes d'analyse syntaxique envisagés.

Au paragraphe précédent un symbole terminal est défini comme une suite de caractères bien fixée (décrit en FACE par une chaîne fermée). On admet maintenant comme terminal, tout mot d'un langage régulier donné.

Le texte postfixé produit par l'analyseur doit refléter le texte source avec toute la précision nécessaire à l'élaboration du programme objet. Bien que les terminaux ne figurent pas explicitement dans la ramification produite, le programmeur FACE pouvait jusqu'alors obtenir toute la précision qu'il jugeait nécessaire grâce aux numéros de règles qu'il affecte lui-même.

Dans la mesure où un terminal n'est plus une chaîne de caractères unique mais un mot d'un certain langage, les numéros de règles n'apportent plus la même précision. Il peut être fort gênant pour compiler un programme de savoir que l'on a rencontré un identificateur, mais de n'avoir aucun moyen permettant de savoir duquel il s'agit !

On pourrait envisager l'insertion du terminal reconnu dans la chaîne postfixée, mais cela risque de l'alourdir assez considérablement et nécessite une nouvelle manipulation et analyse de la chaîne déjà reconnue à chaque passage sur la représentation postfixée, alors que souvent, le programmeur peut prévoir, dès la reconnaissance du terminal, une action, indépendante du contexte syntaxique de l'élément reconnu (mise en table par exemple).

La reconnaissance d'un terminal pourra s'accompagner de l'appel d'une procédure spécialisée qui recevra en paramètre le terminal reconnu. La procédure appelée, dite procédure lexi-

cographique dépend à la fois du type du terminal reconnu, c'est-à-dire du langage régulier auquel il appartient, et de la position de ce terminal au sein de l'arbre syntaxique.

Toute procédure lexicographique reçoit en paramètre le terminal reconnu et rend un résultat qui doit être un pointeur. Ce résultat sera inséré dans la chaîne postfixée avec une marque spéciale indiquant qu'il ne s'agit pas d'un numéro de règle. Au cours de l'exploitation de la chaîne postfixée par APPL, les procédures appelées reçoivent ces résultats en paramètre et peuvent ainsi exploiter les résultats de l'analyse lexicographique (accès en table d'identificateurs par exemple).Dans le cas où des actions sémantiques sont effectuées au cours de l'analyse (procédure ANAPPLYSE), le résultat d'une procédure lexicographique est placé sur la pile des paramètres de la rangée de procédures.

La nécessité des procédures lexicographiques peut être vue sous un autre angle. Jusqu'alors la liste des terminaux fixée par la grammaire était finie. Les numéros de règle permettaient de les reconnaître et de les distinguer tous par une sorte d'adressage calculé implicite dans la table des terminaux. Désormais la liste des terminaux de la grammaire devient infinie. La table des terminaux ne peut donc plus être une liste exhaustive de tous les terminaux possibles. Il faut donc pour chaque programme analysé faire la liste des terminaux utilisés dans ce programme et les identifier. C'est précisément le rôle des procédures lexicographiques.

Pratiquement on est amené à distinguer deux types de symboles terminaux :

- les terminaux définis par une chaîne de caractère.
- les terminaux éléments quelconques d'un certain langage régulier.(Bien que ce type englobe théoriquement le premier, on le distingue pratiquement).

Du point de vue de l'analyse syntaxique aucune distinction n'est faite entre les différents mots d'un même langage régulier (c'est l'affaire de l'analyse lexicographique). A ce

niveau on définira comme un terminal un couple constitué

- d'un langage régulier (sous-entendu : "n'importe quel mot de ce langage")
- une procédure lexicographique.

L'ensemble des procédures lexicographiques utilisées dans une grammaire donnée doit être regroupé dans une rangée de procédures. Cette rangée a pour nom RLEX. Elle est construite automatiquement par le compilateur FACE qui y groupe toutes les procédures référencées comme procédure lexicographique dans une grammaire.

L'introduction de l'analyse lexicographique est une extension du langage FACE. Cela nécessite l'introduction de nouveaux concepts. Dans la définition des grammaires un terminal pourra être

- soit une chaîne fermée, suite de caractères
- soit un identificateur, nom d'un certain langage régulier, suivi du nom de la procédure lexicographique associée placée entre parenthèses. Le nom de la procédure peut éventuellement être vide (ce qui permet de sauter les commentaires par exemple).

Les langages réguliers dont les noms figurent dans les grammaires doivent être définis dans des morphologies. Une morphologie est un nouveau type de déclaration en FACE (au même titre qu'une grammaire).

Dans une morphologie, un langage régulier est défini sous forme d'une expression régulière.

La syntaxe des morphologies est la suivante :

<morphologie>::= morpho <expression factorisée du langage> ;
|<morphologie> <définition de liste de caractères> ;

<définition de liste de caractères>::= <nom de liste> : <caractère>|
<définition de liste de caractères>/<caractère>

<nom de liste>::=<identificateur>

<expression factorisée du langage>::=<identificateur>:<membre alternatif>

<membre alternatif>::=<alternative>/<membre alternatif>|<alternative>|<membre alternatif>/

<alternative>::=<facteur>|<facteur>,<alternative>

<facteur>::=<primaire>|<primaire>*

<primaire>::='<caractère>'|<nom de suite>|(<membre alternatif>)

L'identificateur désignant un langage régulier dans une grammaire doit être l'identificateur d'une<expression factorisée du langage>.

Cette expression est une expression régulière utilisant les opérateurs d'union : / de concaténation : , ainsi que la factorisation : ( ) et l'itération *.

Les caractères du vocabulaire de ce langage peuvent être regroupés dans des listes de caractères que l'on peut référencer par leur nom.

Une règle de déterminisme est imposée : deux alternatives ne doivent pas commencer par les mêmes caractères.

Une souplesse assez grande est laissée à l'utilisateur qui peut utiliser les mêmes morphologies dans des grammaires différentes ou partager des listes de caractères entre plusieurs langages réguliers.

Exemple d'une morphologie :

```
morpho
identificateur : lettre (lettre/chiffre) * ;
lettre : a/b/c/d ;
chiffre : 0/1/2/3/4/5/6/7/8/9 ;
nombre décimal : chiffre, chiffre * ('.', chiffre, chiffre * /) ;
```

On peut associer à chaque langage régulier un automate d'état fini qui le reconnait.

La compilation des morphologies fournit ces automates tels qu'ils sont décrits dans le chapitre suivant.

CHAPITRE III

LES DONNEES DU PREANALYSEUR

Le préanalyseur FADA est appelé par le compilateur FACE après compilation de chaque grammaire rencontrée dans le texte source. Cette compilation a pour objet de transformer la grammaire en un code interne dont la description est l'objet de ce chapitre. Ce code interne constitue l'interface entre le compilateur FACE et le sous-programme FADA. Il est construit à l'intérieur d'une rangée d'éléments, au sens de FACE0, qui est passée en paramètre au préanalyseur. Comme cela a déjà été signalé, on pourra constater que ce code est bien adapté au premier algorithme d'analyse implémenté : l'analyse descendante.

3.1. DESCRIPTION GENERALE

La grammaire est donnée sous forme d'une ramification comme cela était décrit au paragraphe 2.3.

A tout symbole, terminal ou non terminal est associé un numéro d'identification : son NDI. Aux symboles non terminaux sont affectés des NDI négatifs consécutifs de -1 à -n ; aux symboles terminaux des NDI positifs de 0 à t.

La grammaire est représentée par quatre tables :

- la table des non terminaux : TMS (table des méta-symboles)
- la table des symboles terminaux : TDT
- le graphe représentant les règles de la grammaire : GRG
- la table des caractères à sauter : TCS.

Ces tables sont réunies à l'intérieur d'une même rangée d'éléments. Elles sont repérées dans cette rangée par un vecteur de renseignements situé au début de la rangée, et donnant l'adresse et la longueur de chaque table. Les adresses figurant dans le vecteur de renseignements, ainsi que les adresses figurant dans les différentes tables (qui pointent l'une sur l'autre) sont toujours des adresses relatives au début de la rangée. Ainsi la grammaire peut être déplacée en mémoire, rangée sur disque, ramenée à d'autres emplacements sans avoir à se préoccuper de problèmes de translation d'adresses.

3.2. LE VECTEUR DE RENSEIGNEMENTS

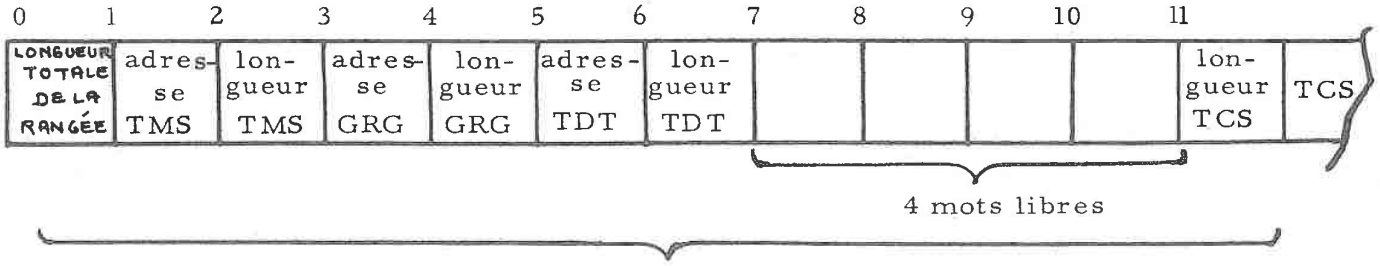
Le premier mot de toute rangée d'éléments contient la taille en mots de la rangée (non compris le premier mot). Compte tenu de ce mot et de la description des 4 tables, FADA peut déduire l'espace disponible en fin de rangée, espace qu'il utilise comme mémoire de travail.

Les six mots suivants du vecteur de renseignements décrivent successivement les tables TMS, GRG et TDT à raison de deux mots par table, le premier donnant son adresse relative, le second sa longueur en mots.

Les quatre mots suivants sont libres. Ils permettent de décrire la table des initiales terminales construite par FADA et communiquée à l'analyseur.

Le dernier mot du vecteur de renseignements donne la longueur de la table des caractères à sauter qui suit immédiatement le vecteur de renseignements.

La structure du vecteur de renseignements est résumée par le schéma ci-dessous :



Il serait possible de simplifier le vecteur de renseignements en ne donnant que les longueurs des tables et en imposant leur ordre d'implantation dans la rangée (ce qui est fait pour la TCS). La solution présentée a été préférée car elle laisse plus de liberté pour la compilation des grammaires.

3.3. LA TABLE DES META-SYMBOLES : TMS

Cette table permet de faire correspondre à chaque non terminal ses règles de réécriture.

La TMS comporte une entrée d'un mot par symbole non terminal de la grammaire. Le rang dans la table du mot correspondant au non terminal dont le NDI est a est -a - 1.

Chaque mot de la table contient un pointeur dans le GRG. Ce pointeur est une adresse relative.

La TMS permet d'accéder à partir du NDI d'un non terminal à la représentation de l'ensemble des membres de droite des règles qui le réécrivent.

3.4. LA TABLE DES TERMINAUX : TDT

La table des terminaux permet d'établir la correspondance entre les NDI des terminaux et leur représentation physique dans le texte source.

Chaque terminal occupe dans la table des terminaux un ou plusieurs mots consécutifs. Le NDI du terminal est le rang dans la TDT du premier mot qui le représente.

o0o

a) Les terminaux chaîne de caractères

La table des terminaux contient la chaîne de caractères (1 caractère par octet). Cette chaîne est suivie d'un ou plusieurs octets à zéro qui marquent la fin du terminal et permettent de le cadrer à droite sur une limite de mot.

Ainsi le terminal >= occupe un mot dans la table des terminaux :

1er octet : >

2ème octet : =

3ème et 4ème octet : zéro.

Le terminal 'début' occupe 2 mots :

'DEB sur le premier mot

UT' sur les 3 premiers octets du 2ème mot et zéro sur le dernier octet.

Le terminal 'AB' occupe deux mots :

'AB' sur le premier mot

zéro sur le second.

o0o

b) Les terminaux définis par un langage régulier (cf. II.10)

L'analyse lexicographique n'est pas implémentée dans la version actuelle de FACE. Ce type de terminal ne fait donc pas partie de l'interface actuellement utilisé, son introduction nécessite une modification du vecteur de renseignements : il faut prévoir deux mots supplémentaires après le mot 6 (contenant la longueur de la TDT).

Les terminaux définis par un langage régulier figureront dans la table des terminaux sous forme d'un couple de deux mots :
le premier mot donne accès au langage régulier,
le second à la procédure lexicographique attachée à ce terminal.
Il est rappelé que ce sont ces deux éléments qui constituent ensemble un terminal ; s'il arrive qu'une autre procédure soit, à un autre endroit de la grammaire, attachée au même langage régulier, il s'agit d'un autre terminal défini ailleurs dans la table des terminaux.

La procédure lexicographique est représentée dans le second mot par son numéro à l'intérieur de la rangée de procédures lexicographiques RLEX, dans laquelle on trouve un mot par procédure (qui contient l'adresse du point d'entrée dans la procédure). Les numéros de procédure vont de 1 à n. Le numéro zéro correspond à la procédure vide (utile pour traiter les commentaires par exemple).

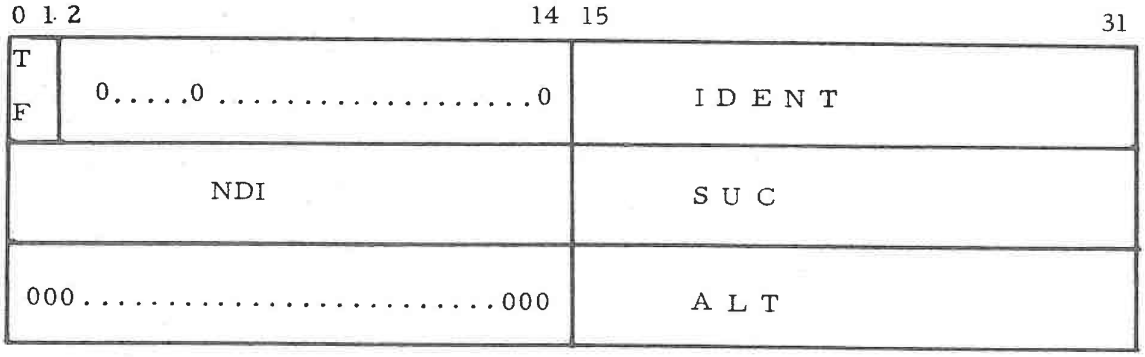
Le langage régulier est représenté par un automate fini déterministe. L'ensemble des automates liés à une grammaire se trouve dans une table spéciale (table morphologique) dont l'adresse et la longueur figurent dans les 2 mots supplémentaires introduits dans le vecteur de renseignements. Le premier mot de la représentation du terminal permet d'accéder à l'état initial de l'automate. Le ler octet de ce mot contient une marque spéciale (zéro par exemple) qui n'est le code d'aucun caractère EBCDIC. Ce ler octet permet de distinguer dans la TDT les deux types de terminaux.

La table morphologique sera décrite ultérieurement.

3.5. LE GRAPHE REPRESENTANT LA GRAMMAIRE : GRG

Le GRG contient le représentation de l'ensemble des membres de droite de la grammaire sous forme d'une ramification binaire les débuts de règles étant factorisés.

Le GRG est constitué de blocs de 3 mots. Chaque bloc référence un symbole ou est une marque de fin de règle. La structure d'un bloc est :



Le champ TF est un drapeau de deux bits qui indique le nature de l'élément représenté dans le bloc.
Si l'élément représenté est un symbole terminal (élément ST), le champ TF a pour valeur ST = B '10'.
Si l'élément représenté est un symbole non terminal (élément SNT) , le champ TF a pour valeur SNT = B '00'.

Si l'élément représenté est une marque de fin de règle (élément FI) le champ TF a pour valeur FI = B '01'.

Le champ SUC contient un pointeur vers l'élément suivant de la règle (adresse de bloc relative au début de la grammaire). Pour un élément ST ou SNT, ce pointeur existe toujours. Il représente le lien suc de 2.4. Dans un élément FI ce champ est toujours à zéro.

Le champ ALT contient éventuellement un pointeur vers une autre alternative. Il représente le lien "alt" du paragraphe 2.4. Le champ ALT des éléments ST ou SNT n'ayant pas de lien alt est mis à zéro. Les éléments FI n'ont pas de lien alt. Le champ ALT de ces éléments est utilisé à une autre fin. Il reçoit le numéro de ligne du programme FACE contenant la règle représentée. Cette information permet au préanalyseur FADA d'émettre les messages d'erreur sur la grammaire en précisant le numéro de la ligne source erronnée.

Les deux liens de l'arborescence binaire sont donc représentés par chaînage. Il aurait été possible de représenter un des liens par contiguité. Cela aurait permis de ramener la taille d'un bloc de trois à deux mots. C'est d'ailleurs ce qui sera fait pour la représentation de la grammaire en vue de l'analyse ascendante. Mais la représentation adoptée à deux liens de chaînage laisse plus de liberté au préanalyseur qui peut aisément modifier la grammaire sans avoir de problème de translation (pour supprimer la récursivité à gauche directe notamment).

Le champ NDI d'un élément ST ou SNT contient le NDI du symbole référencé. Il est nul pour un élément FI.

Le champ IDENT d'un élément ST ou SNT est à zéro (cependant le champ IDENT d'un symbole SNT recevra au cours de son traitement par FADA, l'adresse de ses règles de réécriture dans le GRG, ce qui pourra éviter au cours de l'analyse de passer par la TMS).

Le champ IDENT d'un élément FI contient le numéro de règle. S'il s'agit d'une règle sans numéro, il contient zéro.

3.6 EXEMPLE DE REPRESENTATION D'UNE GRAMMAIRE

Soit la grammaire A ::= BAC ||B 'eg' |2| B

B ::= BC 'abcd' |1| B 'b' |3| C 'efg' |4| 'eg' |5|

C ::= 'gh' |6| 'gh' A || |7| 'vwxyz' |8|

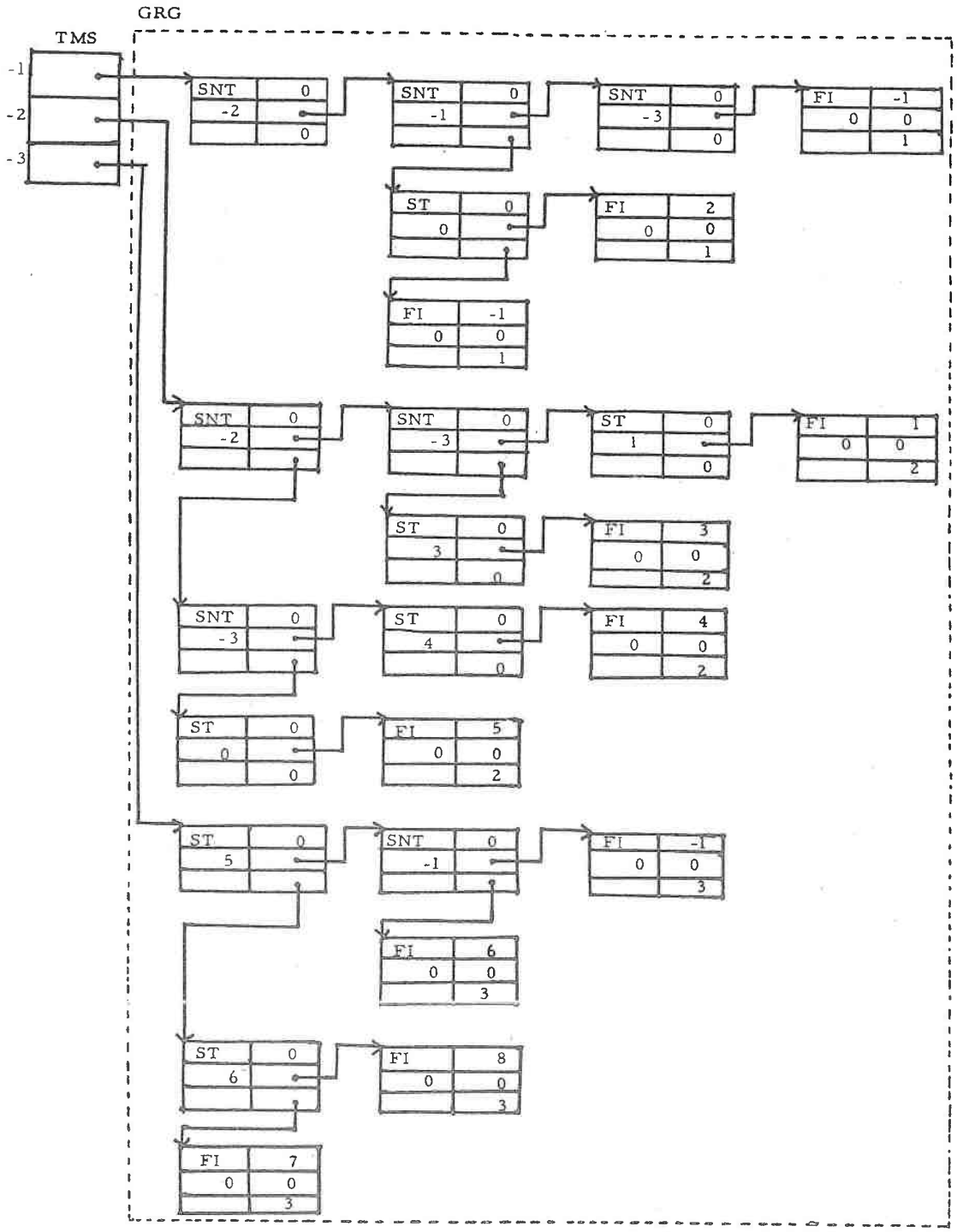
Les NDI des trois symboles non terminaux seront :

-1 pour A, -2 pour B, -3 pour C.

La table des terminaux pourra être

0	e	g	00	00
1	a	b	c	d
2	00	00	00	00
3	b	00	00	00
4	e	f	g	00
5	g	h	00	00
6	v	w	x	y
7	z	00	00	00

La TMS et le GRG sont :


GRG
TMS
-1
-2
-3
SNT 0 -2 0
SNT 0 -1
SNT 0 -3 0
FI -1 0 0 1
ST 0 0
FI 2 0 0 1
FI -1 0 0 1
SNT 0 -2
SNT 0 -3
ST 0 1 0
FI 1 0 0 2
ST 0 3 0
FI 3 0 0 2
SNT 0 -3
ST 0 4 0
FI 4 0 0 2
ST 0 0 0
FI 5 0 0 2
ST 0 5
SNT 0 -1
FI -1 0 0 3
FI 6 0 0 3
ST 0 6
FI 8 0 0 3
FI 7 0 0 3

Les éléments FI n'ont pas de lien "alt". S'il existe un élément FI dans une suite d'alternatives, celui-ci doit se trouver en fin de liste. Cet ordre est celui que doit trouver l'analyseur descendant comme cela est précisé en 2.4.

3.7. LA TABLE DES CARACTERES A SAUTER: TCS

Cette table résulte de la compilation des règles de grammaire sans membre gauche qu'il est possible d'écrire en FACE. Ces règles (facultatives) permettent de spécifier des caractères non significatifs dans le langage à compiler (par exemple les blancs en FORTRAN). Les caractères spécifiés dans ces règles ne sont pas pris en compte lors de l'analyse syntaxique. Ils peuvent apparaître n'importe où dans le texte source. Au cours de l'analyse c'est la procédure de lecture d'un terminal qui exploite la liste des caractères à sauter.

Bien que les caractères à sauter puissent apparaître n'importe où dans le texte source et n'aient pas d'influence sur la signification du langage, le programmeur FACE peut désirer être informé de leur apparition (afin de pouvoir connaître le rang exact d'un caractère dans le texte par exemple). A cette fin, il est possible d'associer à chaque caractère non significatif un numéro de règle. Ce numéro est placé dans la représentation postfixée par l'analyseur chaque fois qu'il rencontre dans le texte source le caractère correspondant.

Dans la TCS chaque caractère non significatif est décrit par un mot contenant :

- dans le 1er octet : zéro
- dans le 2ème octet : le code EBCDIC du caractère
- dans le 2ème demi-mot : le numéro de règle associé (-1 si aucun numéro ne lui est associé).

On peut spécifier en FACE un caractère à sauter spécial (noté $) qui est le passage à un nouvel enregistrement physique dans le texte source (caractère couramment appelé "saut de ligne"). Lorsqu'il figure dans la liste des caractères à sauter d'une grammaire, il y est codé par un octet à zéro.

L'association d'un numéro de règle à ce caractère permet au programme FACE de compter le nombre de lignes du texte source.

Ce caractère est d'une nature différente des autres caractères à sauter car il est créé par la procédure de lecture s'il figure dans la liste des caractères à sauter, mais s'il n'y figure pas, les différents enregistrements sont considérés comme un flot continu sans marque de séparation.

3.8. LA TABLE MORPHOLOGIQUE

La table morphologique permet de représenter l'ensemble des langages réguliers utilisés dans la grammaire sous forme d'automates finis déterministes.

Le vocabulaire terminal est commun à tous les langages réguliers. Il comporte toutes les configurations d'octet pouvant correspondre à un code EBCDIC de caractère imprimable. Plus précisément il s'agit des codes hexadécimaux X'40' à X'FA' inclus, soit 187 symboles.

L'ensemble des automates d'une grammaire constitue une matrice de transition. Les colonnes de la matrice correspondent aux éléments du vocabulaire, soit 187 colonnes. Les lignes correspondent aux différents états des automates. Les éléments de la matrice donnent les transitions, c'est-à-dire le nouvel état de l'automate : si (s, a, s') est une transition alors s' est la valeur de l'élément de la matrice de transition correspondant à la ligne s et à la colonne a. Les colonnes de la matrice sont numérotées de X'40' à X'FA' et c'est directement le code EBCDIC hexadécimal du caractère lu qui détermine la colonne donnant la transition. Chaque élément de la matrice occupe un octet et contient le numéro d'un état. Les états sont donc des entiers compris entre 0 et 255, soit, au plus, 256 états pour tous les automates d'une grammaire (ce qui paraît à priori largement suffisant).

Une colonne supplémentaire (disons la dernière, de numéro X'FB') est réservée pour indiquer les états finaux des divers automates.

L'état 0 est un état particulier commun à tous les automates : l'état d'échec, auquel on aboutit par lecture d'un mot n'appartenant pas au langage reconnu. Cet état n'est pas physiquement représenté par une ligne de la matrice (les transitions à partir de l'état 0 sont (0, a, 0) quel que soit le caractère a).

Les lignes de la matrice sont donc numérotées de 1 à n ($n \leq 255$). La matrice est représentée en mémoire ligne par ligne. Chaque ligne occupe 188 octets, soit 47 mots. Chaque ligne commence donc à une limite de mot.

Si une grammaire comporte les morphologies données en exemple au paragraphe 2.10, sa table morphologique est :

Marque d'état final

		40	41		4A	4B	4C		80	81	82	83	84		EF	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB
	1	0	0	0	0	0	0	0	0	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
(identificateur)	2	0	0	0	0	0	0	0	0	2	2	2	0	0	0	2	2	2	2	2	2	2	2	2	2	0	f
	3	0	0	0	0	4	0	0	0	0	0	0	0	0	0	6	6	6	6	6	6	6	6	6	6	0	
(point)	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	0	
(décimal)	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	0	f
(entier)	6	0	0	0	0	7	0	0	0	0	0	0	0	0	0	6	6	6	6	6	6	6	6	6	6	0	f
(entier pointé)	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	0	f

L'état initial de l'automate "identificateur" est 1.

L'état initial de l'automate "nombre décimal" est 3.

CHAPITRE IV

ETUDE DE LA GRAMMAIRE PAR FADA

ADEQUATION D'UNE ANALYSE DESCENDANTE

4.1 ORGANISATION GENERALE DU PREANALYSEUR

FADA effectue l'étude de la grammaire en vue de réaliser une analyse descendante. Ce n'est que s'il constate l'impossibilité d'une analyse descendante qu'il entreprend une étude de la grammaire en vue de l'analyse ascendante. Cette dernière est décrite au chapitre VI.

L'étude de la grammaire en vue de l'analyse descendante comprend les étapes suivantes :

- suppression de la récursivité à gauche directe (cf. 2.3 et 2.4) ;
- construction des relations J^* et vs ;
- contrôle de la grammaire selon les deux règles de déterminisme (D'1) et (D'2) (cf. 2.3) ;
- si ce contrôle est satisfaisant écriture de la grammaire sur disque sinon étude de la grammaire en vue de l'analyse ascendante.

La suppression de la récursivité à gauche entraîne l'introduction d'éléments de type FI ayant un lien "suc". Le champ SUC des éléments de type FI étant libre il peut être utilisé à cette fin. La zone TF donnant le type de chaque élément du GRG est étendue à 5 bits pour permettre l'introduction des nouveaux types d'éléments et devient le champ TFRE. Les différentes valeurs que peut prendre ce champ sont : ST, SNT, FI, FI1, FI2.

Les règles récursives à gauche $A ::= A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ sont transformées en $A ::= (\beta_1 \mid \ldots \mid \beta_n)\ (\alpha_1 \mid \ldots \mid \alpha_n)^*$.

Un élément de type FI1 est généré à la suite de chaque β_i. Il donne accès par son lien "suc" à la ramification des α_i. Un élément de type FI2 est généré à la suite de chaque α_i. Il donne accès par son lien "suc" à la ramification des α_i (traduction de l'étoile). La distinction entre éléments de type FI1 et FI2 n'est pas utile à l'algorithme d'analyse mais est utile au préanalyseur ainsi qu'à la procédure ANALYSE pour la recherche de l'axiome à partir du numéro de règle.

La suppression de la récursivité à gauche directe est réalisée par le sous-programme 1PASSAGE.

La construction des relations $\mathcal{J}^*$ et vs nécessite la construction des relations $\mathcal{J}$, $\mathcal{F}$ et $\mathcal{S}$. La construction de ces trois relations ne peut se faire que lorsque l'on connait les non terminaux d'où dérive la chaîne vide. Cette recherche est le rôle du sous-programme VIDFINI. Les relations $\mathcal{J}$, $\mathcal{F}$ et $\mathcal{S}$ sont ensuite construites par le sous-programme 2PASSAGE.

Le sous-programme WARSHALL calcule la fermeture transitive de la relation $\mathcal{J}$ ($\mathcal{J}^*$) et la fermeture transitive stricte $\mathcal{F}^+$ de $\mathcal{F}$.

Le sous-programme VØISEVAL calcule la relation vs. De plus il fournit la relation $< = \mathcal{S}\,\mathcal{J}^*$ qui est un intermédiaire de calcul ainsi que la relation $> \mathcal{F}^{-1+}\,\mathcal{S}\,\mathcal{J}^*$ qui se calcule très facilement à ce moment là.

VERIF effectue le contrôle de la grammaire en fonction des règles de déterminisme (D'1) et (D'2). A chaque infraction à l'une de ces règles, il appelle le sous-programme REGLE1 (pour D'1) ou REGLE2 (pour D'2).

Si la grammaire satisfait aux conditions de déterminisme (D'1) et (D'2) elle est placée sur disque, accompagnée de la relation $\mathcal{J}^*$ par le sous-programme ECRIRE, sinon l'étude en vue de l'analyse ascendante est abordée.

La procédure FADA comporte un module de commande dont le rôle consiste à peu près exclusivement à appeler dans le bon ordre les différents sous-programmes énumérés ci-dessus :

FADA: passage des paramètres
appel de INITIAL (initialisations diverses);
appel de 1PASSAGE ;
appel de VIDFINI ;
appel de 2PASSAGE ;
appel de WARSHALL ;
appel de VERIF ;
appel de RAZ (remise à zéro des bits 5 à 14 du premier de chaque

élément du GRG ; un de ces bits est utilisé par le sous-programme VERIF) ;

appel de TESTSAUT (contrôle de la liste des caractères à sauter) ;

si les règles de déterminisme de l'analyse descendante sont vérifiées alors début appel de FIFI2 (mise à zéro du champ ALT des éléments FI et FI2, contenant des numéros de lignes inutiles pour l'analyseur ; les éléments FIl ne contiennent pas de numéro de ligne) ;

appel de FINAL (libération d'espace mémoire et mise en place de la description du tableau des initiaux dans le vecteur de renseignements de la grammaire) ;

appel de ECRIRE ;

appel éventuel du sous-programme TESTONS pouvant faire un listing de la grammaire.

fin

sinon étude de la grammaire en vue de l'analyse ascendante (cf. chapitre VI).

Ces différents sous-programmes vont être décrits sous une forme inspirée du langage ALGOL 60.

4.2. OUTILS UTILISES PAR FADA

a) Les appels de sous-programmes, sauvegardes et restaurations de registres, retour de sous-programmes, se font à l'aide des procédures META-SYMBOL : CALL, SAVE, RESTAUR et RETURN. Ces procédures, décrites en Annexe 1, font partie du SYSTEM SIOUX.

b) Le SYSTEM SIOUX comporte aussi des procédures permettant d'accéder facilement aux différents champs d'un élément du GRG. Ces procédures sont de la forme :

nom de procédure, R1 R2

R2 doit être un registre d'index contenant l'adresse relative de l'élément que l'on désire adresser. Ces procédures sont :

LTFRE : chargement dans le registre R1 de la valeur du champ TFRE ;
LIDENT : chargement dans le registre R1 de la valeur du champ IDENT ;
LNDI : chargement dans le registre R1 de la valeur du champ NDI (avec extension du bit de signe) ;
LSUC : chargement dans le registre R1 de la valeur du champ SUC ;
LALT : chargement dans le registre R1 de la valeur du champ ALT ;
STFRE : rangement du contenu de R1 (les 5 derniers bits) dans le champ TFRE de l'élément dont l'adresse relative est dans R2 ;
SIDENT : rangement du contenu de R1 dans le champ IDENT ;
SNDI : rangement du contenu de R1 dans le champ NDI ;
SSUC : rangement du contenu de R1 dans le champ SUC ;
SALT : rangement du contenu de R1 dans le champ ALT.

Le registre R1 des 5 procédures de rangement doit être un registre pair. Le registre impair immédiatement supérieur doit contenir un masque correspondant au champ adressé. Le système SIOUX définit les masques utiles par des équivalences.

Les procédures de chargement génèrent 3 ou 4 instructions.
Les procédures de rangement génèrent 1 ou 2 instructions.

Dans la suite les accés réalisés par ces procédures sont notés TFRE, IDENT, NDI, SUC, ALT. Ces 5 noms seront utilisés comme des noms de tableaux à une dimension.

c) Le sous-programme PARCØURS est un autre outil important utilisé dans FADA. Ce sous-programme peut parcourir une partie du GRG en appelant un sous-programme à la rencontre de chaque élément.

PARCØURS reçoit en paramètre (ADlELMT) l'adresse du premier élément de la partie du GRG qu'il doit parcourir. PARCØURS passe en revue tous les éléments du GRG que l'on peut atteindre à partir de ce premier élément en suivant les liens "suc" et "alt". L'ordre de parcours se fait en privilégiant le

lien "suc" : suivie de tous les liens "suc" à partir du premier élément jusqu'à un élément FI ou FI2, puis retour en arrière jusqu'à un lien "alt" non nul qui est emprunté etc... Au cours de son travail PARCØURS peut appeler deux sous-programmes :

- un à la rencontre de chaque élément ST, SNT ou FI1
- l'autre à la rencontre de chaque élément FI ou FI2.

Les appels à ces sous-programmes doivent être placés dans le texte de PARCØURS avant son appel au moyen du sous-programme PREPAR qui reçoit les sous-programmes en paramètre. Le sous-programme PREPAR peut être appelé par la procédure ETAP. On peut écrire :

```
ETAP  SP1, SP2
LW,ADIELMT  R5
CALL,PARCØURS
```

Le parcours commence à partir de l'élément dont l'adresse relative est dans R5. PARCØURS appelle le sous-programme SP1 à la rencontre d'un élément ST ou SNT ou FI1 et le sous-programme SP2 à la rencontre d'un élément FI ou FI2.

```
ETAP  ,SP2
```

Aucun sous-programme n'est appelé à la rencontre d'un élément ST, SNT ou FI1 (PREPAR place dans PARCØURS des non opérations au lieu d'appels à SP1).

On peut aussi écrire `ETAP  SP1`

La procédure ETAP fait partie du SYSTEM SIØUX.

Les sous-programmes appelés par PARCØURS reçoivent l'adresse relative de l'élément sur lequel ils sont appelés dans le registre 6.

Le sous-programme PARCØURS effectue donc un cheminement dans un graphe. Il n'utilise aucune zone de mémoire de travail, ni fixe, ni dynamique, mis à part quelques registres (registres 2 à 9). La pile nécessaire à l'algorithme de cheminement est représentée dans le GRG lui-même par inversion des pointeurs "suc" et "alt" ; ce procédé simplifie les problèmes de gestion de mémoire, et permet la réentrance de PARCØURS (les sous-programmes appelés

par PARCØURS peuvent appeler PARCØURS à condition de prendre quelques précautions sur la sauvegarde des registres et de l'appeler sur une partie du GRG disjointe de celle déjà parcourue, (dans laquelle les pointeurs sont inversés).

Le sous-programme PARCØURS s'écrit :

```
début
RPX2 := ADIELMT co RPX2 = paramètre du sous-programme appelé co ;
R4:=[TFRE (RPX2)] ;
si R4 = FI ou R4 = FI2 alors début SPF1 : emplacement d'un appel au 2ème
                                           sous-programme ;
                                  return co retour trivial le 1er élément étant
                                           FI ou FI2 co ;
                               fin ;
emplacement d'un appel au 1er sous-programme ;
RPX1 := RPX2 ;
RPX2 := SUC [ RPX1]  co suivie du lien "suc" co ;
ØNYVA : R4 := TFRE [ RPX2] ;
si R4 = FI ou R4 = FI2 alors allera SPF2 ;
emplacement d'un appel au 1er sous-programme ;
RX3 := SUC [RPX2] co RX3 = adresse de l'élément suivant co ;
SUC [RPX2] := RPX1 co inversion du lien suc co ;
RPX1 := RPX2 ; RPX2 := RX3 ; allera ØNYVA co progression et bouclage co ;
SPF2 : emplacement d'un appel au 2ème sous-programme ;
       co un élément de type FI ou FI2 vient d'être rencontré et traité ;
       PARCØURS revient en arrière en suivant le lien "suc" inversé jusqu'à la
       rencontre d'un lien "alt" ou de l'élément dont il est parti co ;
ØNREVIEN : RX3 := ALT [RPX1] co RPX1 contient l'adresse de l'élément précé-
                dent co ;
           si RX3 = 0 alors allera RØNPØIN ;
           co un lien alt est rencontré ; l'alternative va être parcourue co
              R4 := TFRE [RX3] ;
```

```
si R4 ≠ FI et R4 ≠ FI2 alors début co la nouvelle alternative ne commence pas
                                    par un élément FI ou FI2 co
                                    ALT [RPX1] := RPX2 co mémorisation de
                                    la valeur réelle du lien "suc" dans l'alternative
                                    que l'on quitte ;
                                    le champ suc y reste inversé co ;
                                    RPX2 := RX3 ;
                                    emplacement d'un appel au 1er sous-programme ;
                                    RPX2 := SUC [RX3] ;
                                    SUC [RX3] := RPX1 co inversion du lien "alt"
                                        mémorisé dans le champ suc du 1er
                                        élément de l'alternative co ;
                                    mise en place d'une marque dans l'élément
                                    repéré par RX3 (1er octet du 3ème mot) in-
                                    diquant que l'élément est le 1er d'une alter-
                                    native et que le champ suc pointe vers l'al-
                                    ternative précédente ;
                                    RPX1 := RX3 ; allera ØNYVA co pour un par-
                                    cours de toute l'alternative co ;
                                fin ;
co la nouvelle alternative est réduite à un élément FI ou FI2 co ;
        échange de RPX2 et RX3 ;
        emplacement d'un appel au 2ème sous-programme ;
        échange de RPX2 et RX3 ;
        co la dernière alternative d'une suite d'alternatives est traitée ;
        l'adresse du dernier élément de type ST, SNT ou FIl de cette suite se
        trouve dans RPX1 co
RØNPØIN : si l'élément d'adresse RPX1 n'est pas marqué
            alors co l'élément ne fait pas partie d'une suite alternative ; pas
                arrière normal par le lien "suc" co
              début si RPX1 = AD1ELMT alors return co on est revenu au point
                                    de départ co ;
```

```
                RX3 := SUC [RPX1] ;
                SUC [RPX1] := RPX2 co retablissement du lien"suc" co ;
                RPX2 := RPX1 ; RPX1 := RX3 ; allera ØNREVIEN
            fin ;
        co l'élément repéré par RPX1 est marqué ; il fait donc partie d'une suite
           d'alternatives qu'il faut remonter co
        RX3 := SUC [RPX1] co RX3 : adresse de l'alternative précédente co ;
        SUC [RPX1] := RPX2 co rétablissement du lien"suc"de la dernière alter-
                                        native co ;
ENCØRE : retrait de la marque de l'élément d'adresse RPX1 ;
        RPX2 := ALT [RX3] co RPX2 = véritable lien"suc"de l'élément
                                        d'adresse RX3 co ;
        ALT (RX3) := RPX1 co rétablissement du lien"alt"co ;
        RPX1 := SUC [RX3] ;
        SUC [RX3] := RPX2 co rétablissement du lien"suc"co ;
        échange de RPX1 et RX3 ;
        si l'élément repéré par RPX1 est marqué alors allera ENCØRE ;
        co la première alternative d'une suite d'alternatives est retrouvée
        (repérée par RPX1). RX3 repère l'élément précédent selon le lien"suc".
        Il faut reprendre le retour arrière par le lien"suc"co
        si RPX1 = ADIELM T alors return ;
        RPX2 := RPX1 ; RPX1 := RX3 ; allera ØNREVIEN
        fin .
```

d) Les diverses relations calculées par FADA ($\mathfrak{J}$, $\mathfrak{J}^*$, $\mathfrak{F}$, $\mathfrak{S}$, vs, $\prec$, $\succ$) sont représentées par des matrices booléennes. Pour que ces matrices occupent le minimum de place mémoire, chaque élément de la matrice est un bit. D'autres représentations sont possibles par des techniques de représentations de matrices creuses, mais leur utilité dans ce cas est discutable. En effet, il est assez difficile de rendre compte si ces matrices sont creuses ou non. Cela dépend très largement des grammaires. De plus, il est important, pendant l'analyse d'avoir un accès direct dans ces matrices afin d'atteindre une bonne efficacité. Cette

méthode facilite aussi la gestion de mémoire dans FADA. En effet, la place occupée par ces matrices dépend uniquement du nombre de symboles terminaux et non terminaux de la grammaire. La réservation de chaque matrice peut donc être faite en un bloc avant que son contenu ne soit calculé.

L'utilisation de ces matrices de bits est facilité par 3 procédures regroupées dans le SYSTEM D. Ce sont les procédures DIMENSIØN, SBIT et LBIT.

La procédure DIMENSIØN provoque la réservation dynamique d'une matrice (voir la gestion mémoire au paragraphe suivant) et l'initialise à zéro. Un appel de la procédure s'écrit :

DIMENSIØN, R1, R1+1 T, (A, B), (C, D) [, ADR]

R1 doit être un registre pair. Les registres R1 et R1+1 sont utilisés comme registres de travail.

A, B, C, D peuvent être soit des variables, soit des constantes.

A est l'indice de la première ligne.

B est l'indice de la dernière ligne.

C est l'indice de la première colonne.

D est l'indice de la dernière colonne.

T est le nom de la matrice. Il est utilisé par la procédure DIMENSIØN pour étiqueter un groupe de 4 mots qu'elle génère, destiné à recevoir les renseignements suivants :

1er mot = adresse de mot de la matrice (une matrice de bits commence toujours à une adresse de mot et occupe un nombre entier de mots même si le dernier est incomplétement rempli).

2ème mot = taille en mots de la matrice.

3ème mot = K1 = D - C + 1.

4ème mot = K2 = C * (A-1) - A * (D + 1).

Ces deux dernières constantes permettent de calculer l'adresse relative d'un bit. Le bit d'indices (I, J) se trouve à l'adresse relative de bit :
K1 * I + K2 + J (la matrice est représentée ligne par ligne).

La réservation de la matrice se fait, normalement dynamiquement. Cependant si le 4ème argument est présent, la matrice est réservée à l'emplacement contenu dans ADR.

La procédure SBIT permet de mettre un bit à 1 dans une matrice. Elle s'écrit: SBIT T, (I, J)
mise à 1 du bit ligne I colonne J de la matrice T.

La procédure LBIT permet de charger dans un registre un bit d'une matrice. Elle s'écrit :

LBIT , RX T, (I, J)

RX doit être un registre d'index pair. Il est utilisé comme registre de travail. Chargement du bit de la ligne I colonne J de la matrice T dans le registre RX + 1.

4.3 GESTION DE LA MEMOIRE PAR FADA

FADA reçoit la grammaire dans une rangée d'éléments dont la taille est précisée dans le premier mot. Cette rangée est surdimensionnée.

FADA utilise l'espace disponible à la fin de cette rangée pour y placer les matrices contenant les diverses relations calculées ($\mathfrak{J}$, $\mathfrak{S}$ ...) et comme zone de travail. Cet espace mémoire est alloué et libéré dynamiquement au fur et à mesure des besoins comme une pile. Pour cette raison la rangée contenant la grammaire est utilisée pour construire une pile au sens des instructions de pile de la 10070 (pile MEM). Cette pile est initialisée par le sous-programme INITIAL. Les allocations et libérations de mémoire sont réalisées par des instructions sur pile.

Si des allocations successives conduisent à un débordement de la rangée, il se produit un déroutement sur débordement de pile. Ce déroutement est

récupéré par le programme de gestion de mémoire DEBØRD qui est chargé d'allouer un nouvel espace mémoire.

DEBØRD demande au système l'attribution de nouvelles pages par la procédure SIRIS 7 M:GP. Le nouvel espace obtenu (qui n'est pas contigu au premier) permet d'initialiser un nouvel état de pile dans MEM. L'ancien état de MEM est conservé par DEBØRD dans PILDEPIL. Ainsi l'espace mémoire sur lequel travaille FADA peut être constitué de plusieurs segments disjoints non contigus, appelés "sections de pile". Chaque section est une zone contiguë. En fait il peut y avoir au plus deux sections de pile car les pages allouées par le système au cours de demandes successives sont contiguës.

Si au cours de libérations d'espace, survient un débordement inférieur de la pile MEM, un déroutement se produit et DEBØRD, libère l'espace alloué à la section de pile vide par M:FP et restaure l'état antérieur de la pile MEM.

A la fin de son traitement, après écriture sur disque de la grammaire, FADA libère les pages allouées dynamiquement.

Lorsqu'un débordement supérieur de pile est provoqué par une instruction MSP, demandant l'allocation de plusieurs mots, ces mots sont toujours alloués en une zone contiguë, quitte à ce que DEBØRD laisse des mots inemployés à la fin d'une section de pile. La procédure DIMENSIØN utilise une instruction MSP pour réserver la place d'une matrice, ce qui assure qu'une matrice occupe toujours une zone contiguë de mémoire.

4.4 SUPPRESSION DE LA RECURSIVITE A GAUCHE DIRECTE : lPASSAGE

Les règles récursives à gauche sont détectées facilement par la rencontre dans la liste des premières alternatives des règles de réécriture d'un non terminal, une référence à ce métasymbole. Du fait de la factorisation des règles, cette référence n'existe qu'une fois, et constitue un élément dit élément récursif.

L'élément récursif est retiré de la liste des alternatives, ce qui retire toutes les règles récursives. Tous les éléments FI des règles récursives sont transformées en éléments FI2 dont le lien "suc" pointe vers le successeur de l'élément récursif. Les éléments FI des autres règles de même membre gauche sont transformés en éléments de type FI1 dont le lien "suc" pointe vers le successeur de l'élément récursif. L'élément récursif est enfin transformé en élément FI.

La boucle principale de lPASSAGE est chargée de détecter la récursivité à gauche :

```
début
pour chaque NDI de non terminal faire
début ADPE := adresse du mot de rang - NDI de la TMS ;
      RX1 := (ADPE) co RX1 = adresse relative des règles de réécriture du
                             non terminal NDI co ;
ALT : LTFRE, RP1 RX1 ;
      si RP1 ≠ FI alors début LNDI, RP1 RX1 ;
                          si RP1 = NDI alors ELIMIN
                                sinon début ADPE := adresse du
                                            3ème mot de l'élément
                                            repéré par RX1 ;
                                      RX1 := (ADPE) co RX1
                                             := lien "alt" co ;
                                      si RX1 ≠ 0 alors allera
                                                      ALT
                                fin
                    fin
fin
```

Lorsque la procédure ELIMIN est appelée, RX1 contient l'adresse relative d'un élément récursif.

```
procédure ELIMIN ;
début RP1 := ALT [RX1] ;
      (ADPE) := RP1 co l'élément récursif (et donc toutes les règles récursi-
                ves) est retiré de la liste des alternatives co ;
```

```
      ETAP   ,BØUCLAGE co préparation d'un appel de PARCØURS pour la
                              construction des éléments FI2 co ;
      ADIELMT := SUC [RX1]  co point de départ de PARCOURS = successeur
                              de l'élément récursif co ;
      RECUR := ADIELMT  co RECUR contient ce qui détiendra le lien "suc"
                              des éléments FI1 et FI2 co ;
      PARCØURS ;
      co Les éléments FI2 sont construits co
      construction d'un élément FI sans numéro de règle (-1) à la place de
      l'élément récursif (d'adresse RX1);
      RX2 := RECUR ;
CHERFI : RX2 := ALT [RP1] ;
         si RP1 ≠ 0 alors début RX2 := RP1 ; allera CHERFI fin ;
         ALT [RX2]  := RX1  co l'élément FI construit est placé en alterna-
                              tive des parties de règles récursives co ;
         ETAP    ,ENTREE  co le sous-programme ENTREE doit construire
                              les éléments de type FI1 co ;
         ADIELMT := TMS [-NDI] ;
         si  ADIELMT ≠ 0  alors  PARCØURS
                          sinon  co le retrait des règles récursives a pro-
                                 voqué le rangement de la valeur zéro dans
                                 la TMS ; cela signifie que toutes les règles
                                 de réécritures de ce symbole sont récur-
                                 sives à gauche, ce qui est normal co
                          début  préparation et impression d'un message
                                 d'erreurs  co un numéro de ligne est recher-
                                 ché grâce à un appel de PARCØURS avec
                                 RECUR en paramètre. Le sous-programme
                                 NØLIGNE appelé par PARCØURS à la ren-
                                 contre d'un élément FI2 extrait le numéro
                                 de ligne lors de son premier appel puis exé-
                                 cute un ordre ETAP sans argument de
```

manière à ce que PARCOURS se termine
sans l'appeler à nouveau co ;
un élément FI1 sans numéro de règle est
généré à la suite du GRG. Son successeur
est RECUR ;
TMS [-NDI] pointe sur cet élément FI1
co cela revient à conserver la récursivité
mais à ajouter une règle vide co

fin

fin

Les sous-programmes BOUCLAGE et ENTREE reçoivent dans le registre 6 l'adresse d'un élément FI. Cet élément FI est transformé en élément FI2 (pour BOUCLAGE) ou FI1 (pour ENTREE). Le lien "suc" de cet élément est RECUR. ENTREE met à zéro le troisième mot.

4.5. RECHERCHE DES REGLES VIDES : VIDFINI

Le but de VIDFINI est de noter tous les non terminaux d'où peut dériver la chaîne vide. Ces non terminaux sont provisoirement mémorisés par marquage du premier octet de l'entrée correspondante de la TMS. Cette marque sera reportée dans la dernière colonne de la table des initiaux terminaux par 2PASSAGE (cette table n'est pas encore réservée au moment de l'appel de VIDFINI).

L'algorithme utilisé par VIDFINI consiste à construire l'ensemble des membres de gauche des règles vides, puis à y ajouter les symboles qui se réécrivent par des symboles de l'ensemble et ainsi de suite jusqu'à obtenir deux fois de suite le même ensemble. Le nombre d'itérations sera au plus égal au nombre des non terminaux de la grammaire.

Dans un premier temps VIDFINI construit l'ensemble des symboles

membres gauches d'une règle vide. Ces symboles sont marqués dans la TMS.

Ensuite VIDFINI marque les symboles membres gauches d'une règle ne comportant que des symboles déjà marqués.

Cette opération est répétée jusqu'à ce qu'elle ne provoque plus de nouveaux marquages.

L'examen des règles comportant des symboles marqués est assuré par le sous-programme TØUVID.

TØUVID suit les règles à partir d'un élément du GRG qui lui est donné en paramètre jusqu'à la rencontre :

- d'un élément FI, FI1 ou FI2
- d'un symbole terminal
- d'un symbole non terminal non marqué dans la TMS.

Comme PARCØURS, TØUVID peut appeler des sous-programmes spécifiques à la rencontre de chaque élément. Les appels à ces sous-programmes sont placés dans TØUVID par le sous-programme PRTØUVID.

PRTØUVID a 4 instructions en paramètre :

- l'instruction qui sera exécutée à la rencontre d'un élément FI (I1)
- l'instruction qui sera exécutée à la rencontre d'un élément FI2 (I2)
- l'instruction qui sera exécutée à la rencontre d'un élément FI1 (I3)
- l'instruction qui sera exécutée à la rencontre d'un élément ST ou SNT (I4).

Lorsque ces instructions sont exécutées, le registre 5(EXAMIN) contient l'adresse relative de l'élément et lorsqu'il s'agit d'un élément ST ou SNT, le registre 4 (RX1) contient le NDI.

Selon la valeur placée dans la variable BØØLTØU, il est possible de demander à TØUVID de poursuivre l'examen des règles au delà d'un élément FI1 (BØØLTØU=1) ou de s'arrêter sur un élément FI1 (BØØLTØU=0).

L'algorithme utilisé par TØUVID pour parcourir le GRG est plus simple que celui utilisé par PARCØURS. Il utilise le fait que, au moment où TØUVID est appelé le champ IDENT des symboles terminaux et non terminaux n'est pas rempli. Ce champ est utilisé pour construire une liste chaînée des alternatives à parcourir ultérieurement. La tête de liste se trouve dans une variable de TØUVID : TETEV.

```
procédure TØUVID ;
début TETEV := 0 ;
VØYØNS : EXAMIN := TFRE [RX1]   co EXAMIN = adresse relative de
                                   l'élément examiné  co ;
         si RX1 = FI  alors  début  emplacement d'une instruction I1 ;
                                    allera  SORTIR
                             fin ;
         si RX1 = FI2 alors début  emplacement d'une instruction I2 ;
                                    allera  SORTIR
                             fin ;
         si RX1 = FI1 alors  début  emplacement d'une instruction I3 ;
                                    si BOOLTOU  alors  début  EXAMIN :=
                                          SUC [EXAMIN]  ;
                                                allera VOYONS
                                                fin
                                       sinon  allera  SORTIR
                             fin ;
         co EXAMIN pointe sur un élément ST ou SNT co
VOYONS1: RX1 := NDI [EXAMIN]  ;
         emplacement d'une instruction I4 ;
         si  RX1 < 0 et  TMS  [RX1]   est marqué alors
                  co l'élément rencontré est un non terminal d'où peut
                     dériver la chaîne vide  co
```

```
début  RX1 := EXAMIN ;
              EXAMIN = ALT [RX1]  ;
       si EXAMIN ≠ 0 alors co avant d'examiner le succes-
              seur du non terminal d'où dérive la chaîne
              vide, on chaîne son alternative dans TETEV
              s'il s'agit d'un symbole ou on la traite im-
              médiatement s'il s'agit d'un élément FI, FI1
              ou FI2 (dont le champ IDENT n'est pas libre)
              co
début  LA : R2 := TFRE [EXAMIN]  ;
       si R2 = FI alors  début  emplacement d'une instruc-
                                tion I1 ;
                                allera  SUITE
                        fin
       si R2 = FI2 alors  début  emplacement d'une ins-
                                 truction I2 ;
                                 allera  SUITE
                         fin
       si R2 = FI1 alors  début  emplacement d'une ins-
                                 truction I3 ;
                                 si BOOLTOU alors début
                                     EXAMIN := SUC
                                               [ EXAMIN]
                                      allera LA
                                      fin ;
                         allera SUITE
                         fin ;
       co l'alternative est un élément ST ou SNT qui va
       être placé dans la liste TETEV  co
       IDENT [EXAMIN]  := TETEV ;
       TETEV := EXAMIN
fin ;
```

```
                    co les alternatives de l'élément d'adresse (RX1) sont
                    traitées ou mémorisées. On passe à son successeur
                    co
          SUITE :   EXAMIN := SUC [ RX1 ] ;
                    allera VOYONS
          fin ;
          co L'élément rencontré est soit un symbole terminal, soit un
          symbole non terminal d'où ne peut dériver la chaîne vide. On ne
          passe pas au successeur co
                    EXAMIN := ALT [EXAMIN] ;
          si EXAMIN ≠ 0 alors allera VOYONS ;
SORTIR : si TETEV ≠ 0 alors début EXAMIN := TETEV ;
                                  TETEV := IDENT [EXAMIN] ;
                                  IDENT [EXAMIN] := 0 ;
                                    allera VOYONS1
                              fin
fin
```

Le sous-programme VIDFINI appelle le sous-programme TOUVID en lui faisant parcourir l'ensemble des règles de réécriture de chaque non terminal. Si TOUVID atteint un élément FI ou FI1 le non terminal est marqué. Cette opération est répétée juqu'à ce qu'elle ne provoque plus de nouveau marquage.

La première fois que TOUVID est appelé, aucun symbole n'est marqué. TOUVID détecte donc seulement les règles vides.

VIDFINI s'écrit :

```
début  BOOLTOU = 0 ;
       PRTOUVID (MARQUAGE, ERREUR, MARQUAGE, non opération) ;
       co il est normalement impossible de rencontrer un élément FI2 sans
       rencontrer auparavant un élément FI1 ; le sous-programme ERREUR
       est donc un moyen de contrôle de la codification de la grammaire co
DEB :  ENCORE := 0  co ENCORE passe à 1 lorsque des non terminaux sont
                       marqués co ;
```

pour chaque non terminal NDI faire
si TMS [-NDI] non marqué alors début EXAMIN := TMS [-NDI] ;
TOUVID
fin ;
si ENCORE ≠ 0 alors allera DEB
fin

Le sous-programme MARQUAGE consiste simplement à marquer TMS [-NDI] et à positionner la variable ENCORE à 1.
En fait, une fois que le non-terminal est marqué, il est inutile que TOUVID poursuive son travail. Aussi MARQUAGE détruit-il la liste chaînée éventuellement construite par TOUVID, dont le pointeur de tête est TETEV, puis MARQUAGE se débranche à la suite de l'appel de TOUVID.

Remarque : Certaines des possibilités de TOUVID qui peuvent paraître inutiles ici, sont utilisées par 2PASSAGE (voir paragraphe suivant).

4.6. CONSTRUCTION DES RELATIONS $\mathcal{J}$, $\mathcal{S}$ et $\mathcal{F}$: 2PASSAGE

Avant de calculer les relations, le premier rôle de 2PASSAGE est de réserver les matrices booléennes qui les contiendront. Ces matrices sont bien entendu réservées grâce à la procédure DIMENSION.

La relation $\mathcal{J}$ est représentée sur deux matrices : la matrice IN des initiales non terminales et la matrice INT des initiales terminales. En effet après calcul de la fermeture transitive de la relation (calcul qui se fait sur les matrices IN et INT elles-mêmes) seules les initiales terminales sont intéressantes, aussi bien au niveau de FADA que de l'analyseur descendant. La matrice IN pourra donc être détruite et son espace mémoire utilisé à d'autres fins. La dernière colonne de INT sert à noter les non terminaux d'où dérivent la chaîne vide.

L'ordre de réservation des différentes matrices n'est pas quelconque.

Les matrices qui devront être détruites les premières sont réservées en dernier de manière à permettre la gestion en pile de l'espace mémoire.

Soient LTMS le nombre de non terminaux de la grammaire et LTDT son nombre de terminaux (le dernier terminal a pour NDI : LTDT-1). Les réservations faites par 2PASSAGE sont :

DIMENSION, R1, RI1	INT, (-LTMS, -1), (0, LTDT)
DIMENSION R1, RI1	IN, (-LTMS, -1), (-LTMS, -1)
DIMENSION R1, RI1	FIN, (-LTMS, -1), (-LTMS, LTDT-1)
DIMENSION, R1, RI1	PRECED1, (-LTMS, -1), (0, 3*LTDT)
DIMENSION, R1, RI1	PRECED2, (0, LTDT), (0, 2*LTDT)
DIMENSION R1, RI1	VDR, (-LTMS, LTDT-1), (-LTMS, LTDT-1)
DIMENSION, R1, RI1	VDRFI2, (0, 0), (-LTMS, LTDT-1)

FIN est la matrice contenant la relation $\mathcal{F}$

VDR est la matrice contenant la relation $\mathcal{S}$.

PRECED1 et PRECED2 sont des matrices de précédence qui sont réservées à cet endroit pour des raisons de gestion de mémoire, mais qui ne seront remplies que par VOISEVAL (paragraphe 4.8).

VDRFI2 est un vecteur booléen de travail, utilisé par 2PASSAGE et détruit à la fin de ce sous-programme.

2PASSAGE utilise largement les possibilités du sous-programme TOUVID.

TOUVID permet en effet,

lorsqu'il est appelé à partir du début des règles de réécriture d'un non terminal A, d'atteindre tous les éléments ST ou SNT, B tels que $A ::= \varphi\ B\ \psi$ avec $\varphi \overset{*}{\longrightarrow} \wedge$, c'est-à-dire tous les B tels que $B\ \mathcal{J}\ A$,

lorsqu'il est appelé à partir d'un élément ST ou SNT, B des règles de réécriture de A,

d'une part d'atteindre tous les éléments C tels que $A ::= \varphi\ B\ \mu\ C\ \Psi$ avec $\mu \overset{*}{\longrightarrow} \wedge$, c'est-à-dire tous les éléments C tels que $B\ \mathcal{S}\ C$,

d'autre part s'il atteint un élément FI, FI1 ou FI2 alors $A ::= \varphi\ B\ \psi$ avec $\psi \overset{*}{\longrightarrow} \wedge$ c'est-à-dire $A\ \mathcal{F}\ B$.

L'organisation générale de 2PASSAGE consiste à utiliser TOUVID pour construire les initiales de chaque non terminal, puis à appeler PARCOURS sur l'ensemble des règles de réécriture du non terminal. Le sous-programme CONST appelé par PARCOURS est chargé de construire les relations $\mathcal{S}$ et $\mathcal{F}$. Il utilise TOUVID.

```
début BOOLTOU = 1 ;
      ETAP   CONST ;
      MODIFIE := 0  co variable utilisée par CONST  co ;
      pour chaque NDI de non terminal faire
      début PRTOUVID (, , , INITIALE) co INITIALE sera appelé par TOUVID
                                         à la rencontre d'un élément ST ou
                                         SNT co ;
            EXAMIN := AD1ELMT := TMS [-NDI]  ;
            TOUVID  co  construction des initiales co ;
            PRTOUVID (TRAITFI, TRAITFI2, , TRAITSYM) ;
            PARCOURS co construction des successeurs et des finales  co ;
            remise à zéro du vecteur VDRFI2 utilisé par CONST
      fin ;
restitution de l'espace utilisé par le tableau VDRFI2 (utilisé par CONST);
recopie des marques de la TMS dans la dernière colonne de la matrice INT et
suppression de ces marques dans la TMS,
si MODIFIE ≠ 0 alors début TFRE [ MODIFIE ] := TYPE  co  voir explication
                                   dans le sous-programme  CONST  co
                        MODIFIE:=0
                     fin
fin
```

Le sous-programme INITIALE reçoit dans le registre R4 le NDI d'un élément ST ou SNT rencontré par TOUVID. Il ne lui reste qu'à remplir la matrice IN ou INT :

```
procédure INITIALE ; si  R4 < 0 alors   IN [NDI , R4]  := 1 ;
                                sinon   INT [NDI, R4]  := 1 ;
```

Le sous-programme CONST reçoit dans le registre R6 l'adresse d'un élément ST, SNT ou FI1.

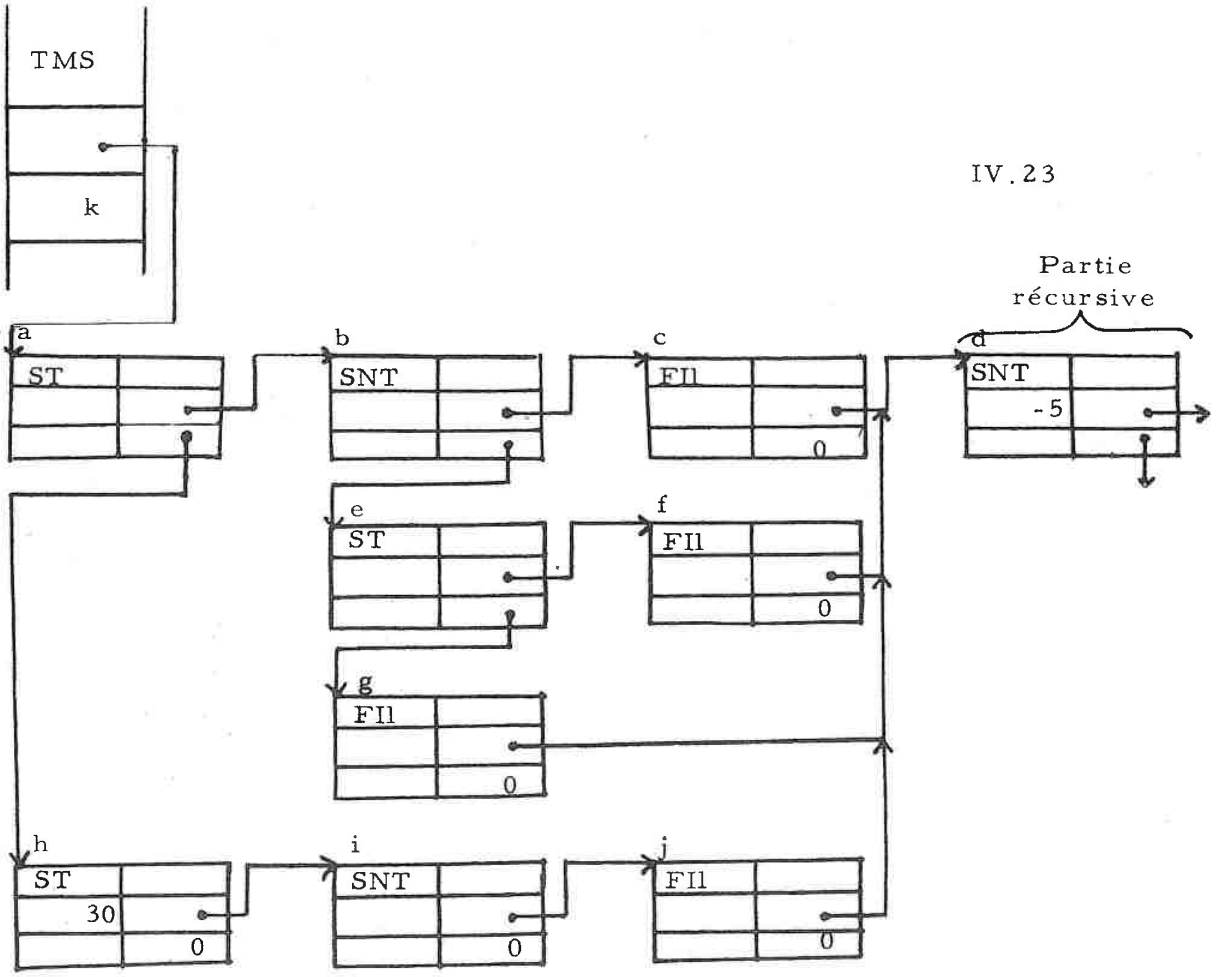
Lorsque l'élément rencontré est ST ou SNT, CONST appelle TOUVID sur le successeur de l'élément afin de remplir la matrice VDR et éventuellement la matrice FIN si TOUVID parvient à un élément FI, FI1 ou FI2.

Lorsque l'élément rencontré par CONST est un élément FI1, CONST recherche ses successeurs (grâce à TOUVID) et les note dans le vecteur VDRFI2.

Lorsque ultérieurement, un élément ST ou SNT a pour successeur un élément FI2, CONST affecte comme successeurs à cet élément tous ceux qui sont notés dans VDRFI2.

Tous les éléments FI1 figurant dans les règles d'un même membre de gauche ont le même successeur (qui est la partie récursive). La partie récursive est donc, en principe parcourue plusieurs fois, ce qui ici est tout à fait inutile. Pour éviter cela, CONST utilise le procédé suivant.

Lorsque l'élément rencontré est du type FI1, CONST examine son successeur. Si cet élément n'a pas été marqué précédemment il reçoit une marque (dans la zone inutilisée de son premier mot). Si l'élément avait été marqué précédemment son adresse relative et son type sont notés dans les variables MODIFIE et TYPE, puis son type est transformé temporairement en type FI. Cela oblige PARCOURS à revenir en arrière sans passer à nouveau sur la partie déjà traitée. Au prochain appel de CONST le type normal est rétabli et ne sera plus modifié si 2PASSAGE est passé à l'élément suivant de la TMS. Si l'on est toujours sur le même poste de la TMS, le sous-programme TOUVID peut passer sur cet élément en le traitant correctement puisque son type est restauré, mais la marque subsiste. La rencontre d'un nouvel élément FI1 par CONST provoquera donc une nouvelle transformation en type FI. Exemple :



A l'appel de CONST le registre 6 prend successivement les valeurs a, b, c. Lorsque R6 = c, l'élément traité est FIl. Son successeur n'est pas marqué. Donc CONST le marque et rend le contrôle. R6 prend la valeur d puis toutes les adresses des éléments de la partie récursive qui est alors traitée. R6 prend alors les valeurs e puis f. L'élément traité est un élément FIl dont le successeur est marqué. Le type de d est conservé dans TYPE, son adresse dans MODIFIE (TYPE = SNT ; MODIFIE = d) avant d'être transformé en élément FI.

Le registre 6 prend la valeur g. MODIFIE est différent de 0. Le type SNT contenu dans TYPE est placé en d. Mais l'élément d'adresse g est de type FIl. Son successeur est marqué, le type SNT et l'adresse d sont aussitôt rangés dans TYPE et dans MODIFIE. Un type FI est placé en d. La valeur suivante du registre 6 est donc h. MODIFIE étant $\neq 0$, le type SNT est replacé en d. MODIFIE est remis à zéro. Cette opération est bien nécessaire car si le successeur de l'élément d'adresse h est un non terminal d'où peut dériver la

chaîne vide, il est nécessaire d'établir une relation de succession ($\mathcal{S}$) entre le terminal 30 et le non terminal -5.

Lorsque le registre 6 prend les valeurs i et j MODIFIE est à zéro. L'élément d'adresse j est de type FIl. Son successeur est marqué. L'élément d'adresse d est donc transformé en élément FI. Son type SNT et son adresse sont mémorisés dans TYPE et MODIFIE.

Le sous-programme PARCOURS rend le contrôle puis est appelé à nouveau par 2PASSAGE avec l'adresse k en paramètre. PARCOURS appelle à nouveau CONST après avoir placé k dans R6. MODIFIE étant différent de zéro, le type SNT est rétabli à l'adresse d ; MODIFIE est remis à zéro. Le type de l'élément d'adresse d ne sera plus modifié.

Il est évident que lorsque la boucle générale sur la table des non terminaux est terminée, 2PASSAGE doit tester la valeur de MODIFIE et éventuellement rétablir le dernier type modifié par CONST.

o0o

Il est maintenant possible d'écrire le sous-programme CONST.

```
procédure CONST ;
début si MODIFIE ≠ 0 alors début  TFRE [ MODIFIE ]  := TYPE
                                  MODIFIE := 0
                             fin ;
     R1:= TFRE [ R6 ]  co R1 = type de l'élément traité co ;
     si R1 = FIl alors début R6:= SUC [ R6]  co R6 = successeur de l'élément
                                                     FIl  co
                             si l'élément d'adresse R6 est marqué alors
                                          début     MODIFIE := R6 ;
                                                    TYPE := TFRE [R6]  ;
                                                    TFRE [R6]  := FI ;
                                                    Return
                                          fin ;
```

```
                co R6 = adresse de début d'une partie récursive
                non encore traitée co
                l'élément d'adresse (R6) est marqué ;
                co les successeurs de l'élément FI1 sont aussi les
                successeurs des éléments FI2 qui se trouvent à la
                fin de la partie récursive. Tous ces successeurs
                sont maintenant mémorisés dans le tableau VDRFI2
                co
                PRTOUVID (,,, VDRFI2 [ 0, R4 ] := 1)
                EXAMIN := R6 co paramètre de TOUVID co ;
                TOUVID ;
                PRTOUVID (TRAITFI, TRAITFI2,, TRAITSYM) ;
                Return
            fin ;
        co CONST a reçu en paramètre un élément de type ST ou SNT co
        NDI6 := R1 := NDI de l'élément traité ;
        EXAMIN := SUC [R6]  co EXAMIN := successeur de l'élément
                                := paramètre de TOUVID co ;
        TOUVID  co construction des relations S (grace aux sous-programmes
                TRAITSYM et TRAITFI2) et F (grace aux sous-programmes
                TRAITFI et TRAITFI2) co ;
        si NDI6 < 0 alors co construction du lien IDENT des non terminaux co
                        IDENT [NDI6] := ADIELMT co ADIELMT paramètre
                                                de PARCOURS co
fin.
```

o0o

Le sous-programme TRAITFI est appelé à la rencontre d'un élément FI par TOUVID. Le symbole NDI6 de CONST est une finale du non terminal membre de gauche actuellement étudié par 2PASSAGE.

```
procédure TRAITFI ; FIN [NDI, NDI6] := 1 ;
```

o0o

Le sous-programme TRAITSYM est appelé par TOUVID à la rencontre d'un symbole terminal ou non terminal. Ce symbole dont le NDI est dans R4 est un successeur du symbole reçu en paramètre par CONST.

```
procédure TRAITSYM ; VDR [NDI6, R4] := 1 ;
                        co R4 est le NDI du symbole rencontré par TOUVID co
```

o0o

Le sous-programme TRAITFI2 est appelé par TOUVID à la rencontre d'un élément FI2. L'effet de TRAITFI2 est le même que TRAITFI, mais en plus les successeurs de l'élément FI2 (mémorisés dans VDRFI2) sont affectés comme successeurs au symbole paramètre de CONST (NDI6).

```
procédure TRAITFI2 ;
début pour i:= - LTMS pas 1 jusqua LTDT -1 faire
      si VDRFI2 [0,i] = 1 alors VDR [NDI6,i] := 1 ;
      TRAITFI
fin.
```

4.7. CONSTRUCTION DES FERMETURES TRANSITIVES $\mathfrak{J}^*$ ET $\mathfrak{F}^*$: WARSHALL

La fermeture transitive des relations $\mathfrak{J}$ et $\mathfrak{F}$ s'élabore à l'aide de l'algorithme de WARSHALL [16,17] .

Le sous-programme construit la fermeture transitive au sens large de la relation $\mathfrak{J}$ car c'est la seule nécessaire. La relation transitive stricte de la relation $\mathfrak{F}$ est nécessaire au calcul de la relation $\succ$. La relation $\mathfrak{F}^*$ s'en déduit trivialement. Il tient compte aussi du fait que les terminaux ne sont origine d'aucun arc.

```
procédure WARSHALL ;
début pour i:= - LTMS pas 1 jusqua -1 faire
      début pour j:= - LTMS pas 1 jusqua -1 faire
            début si IN [j,i] alors
```

```
                    début pour k := -LTMS pas 1 jusqua -1 faire
                          si IN[i,k] alors IN[j,k] := 1 ;
                          pour k := 0 pas 1 jusqua LTDT -1 faire
                          si INT [i,k] alors INT [j,k] := 1 ;
                    fin ;
               si FIN [j,i] alors
                    pour k := - LTMS pas 1 jusqua LTDT -1 faire
                    si FIN [i,k] alors FIN [j,k]
          fin ;
          IN [i,i] := 1
     fin
fin
```

4.8. CALCUL DES RELATIONS DE PRECEDENCE DE LA RELATION vs : VOISEVAL

Les matrices de précédence destinées à recevoir les relations $\lessdot$ et $\gtrdot$ ont été réservées par le sous-programme 2PASSAGE.

La matrice PRECED1 a pour dimensions : [-LTMS : -1, 0 : 3 * LTDT] Elle est destinée à recevoir les relations de précédence entre les non terminaux et les terminaux. Elle contient 3 bits par élément. La dernière colonne est remplie par l'analyseur dans une phase d'initialisation (elle dépend de l'axiome). Le sous-programme VOISEVAL ne remplit que les 2 derniers bits de chaque élément. Le premier sera utilisé par le préanalyseur ascendant (cf. chapitre 6). Le dernier bit sert à mémoriser la relation $\lessdot = \mathcal{F}\mathcal{J}^*$, le bit précédent mémorise la relation $\mathcal{F}^{-1+}\ \mathcal{F}\mathcal{J}^*$ (c'est la relation $\gtrdot$ compte non tenu de son extension aux règles vides). A la sortie de VOISEVAL, pour tout couple (A, t) A $\in$ N, t $\in$ T, PRECED1 [A,t] peut donc avoir les valeurs suivantes :

PRECED1 [A,t] = 0 $\Longleftrightarrow$ pas de relation de précédence entre A et t

PRECED1 [A,t] = 1 $\Longleftrightarrow$ $A \lessdot t \Longleftrightarrow A \mathcal{S}^{*} t$

PRECED1 [A,t] = 2 $\Longleftrightarrow$ $A \mathcal{S}^{-1} + \mathcal{S}\mathcal{J}^{*} t \mathcal{S} \Longrightarrow A \gtrdot t$

PRECED1 [A,t] = 3 $\Longleftrightarrow$ il existe un conflit de précédence entre A et t.

La matrice PRECED2 a pour dimension : [0 : LTDT, 0 : 2 * LTDT] . Elle est destinée à recevoir les relations de précédence entre les symboles terminaux. Elle ne nécessite que deux bits par élément. C'est cette différence de taille entre les éléments qui a conduit à séparer la matrice de précédence en deux, afin d'économiser l'espace mémoire. La dernière ligne et la dernière colonne dépendent de l'axiome de la grammaire et sont donc remplies par l'analyseur. Le premier bit sert à mémoriser la relation $\lessdot = \mathcal{S}\mathcal{J}^{*}$, le second la relation $\mathcal{S}^{-1} + \mathcal{S}\mathcal{J}^{*}$. A la fin de VOISEVAL, pour tout couple (t1, t2) $\epsilon T \times T$ PRECED2 [t1, t2] peut avoir les valeurs suivantes :

PRECED2 [t1,t2] = 0 $\Longleftrightarrow$ pas de relation de précédence entre t1 et t2

PRECED2 [t1,t2] = 1 $\Longleftrightarrow$ $t1 \mathcal{S}\mathcal{J}^{*} t2 \Longleftrightarrow t1 \lessdot t2$

PRECED2 [t1,t2] = 2 $\Longleftrightarrow$ $t1 \mathcal{S}^{-1} + \mathcal{S}\mathcal{J}^{*} t2 \Longrightarrow t1 \gtrdot t2$

PRECED2 [t1,t2] = 3 $\Longleftrightarrow$ conflit de précédence entre t1 et t2.

Dans une première partie, VOISEVAL construit la relation $\lessdot$ par produit de la matrice VDR avec la matrice INT :

```
pour j := 0 pas 1 jusqua LTDT - 1 faire
début pour i := - LTMS pas 1 jusqua -1 faire
      si VDR [i,j] alors PRECED1 [i, 3 * j + 2] := 1
                  sinon début pour k := - LTMS pas 1 jusqua -1 faire
                              si VDR [i,k] et INT [k,j] alors
                                 début PRECED1 [i, 3 * j + 2] := 1 ;
                                       allera fin1
                                 fin ;
                        fin 1 : fin ;
pour i := 0 pas 1 jusqua LTDT -1 faire
si VDR [i,j] alors PRECED2 [i, 2 * j + 1] := 1
            sinon début pour k := - LTMS pas 1 jusqua -1 faire
```

```
                    si  VDR [i,k]  et INT [k,j]  alors
                        début PRECED2 [i, 2 * j + 1]  := 1 ;
                              allera  fin2
                        fin ;
              fin2 : fin
        fin ;
```

La dernière partie de VOISEVAL construit les relations $\mathcal{F}^{-1} + \mathcal{F}\,\mathcal{J}^*$ et vs par produit de la matrice FIN transposée et des matrices PRECED1 et PRECED2. La relation vs est placée dans la matrice VOISIN qui est réservée dynamiquement par VOISEVAL. La relation vs est nécessaire pour effectuer le contrôle de la grammaire en vue de l'analyse descendante (règle de déterminisme (D'2)). Il n'est utile que de connaître les voisins terminaux des symboles non terminaux. La dimension de la matrice VOISIN est donc : [-LTMS : -1, 0 : LTDT - 1] .

```
pour j := 0 pas 1 jusqua LTDT -1 faire
début pour i := - LTMS pas 1 jusqua -1 faire
      début pour k := - LTMS pas 1 jusqua -1 faire
            si FIN [k,i]  et PRECED1 [k, 3 * j + 2]  alors
                  début  PRECED1 [i, 3 * j + 1]  := VOISIN [i,j]  := 1 ;
                        allera  fin3
                  fin ;
            si PRECED1 [i,  3 * j + 2]  alors VOISIN [i,j]  := 1  co  car
            FIN contient la relation F+ et non F* nécessaire au calcul de
            vs co ;
      fin3 : fin ;
      pour i := 0 pas 1 jusqua LTDT -1 faire
      début pour k := - LTMS pas 1 jusqua -1 faire
            si FIN [k,i]  et PRECED1  [k, 3 * j-1]  alors
                  début PRECED2 [i, 2 * j]  := 1 ;
                        allera fin4
                  fin ;
      fin4 : fin
fin
```

4.9. CONTROLE DE LA GRAMMAIRE EN VUE D'UNE ANALYSE DESCENDANTE : VERIF

Le but du sous-programme VERIF et de relever toutes les infractions aux règles de déterminisme (D'1) et (D'2).

A chaque infraction à la règle (D'1) VERIF appelle le sous-programme REGLE1.

A chaque infraction à la règle (D'2) VERIF appelle le sous-programme REGLE2.

Plusieurs versions des sous-programmes REGLE1 et REGLE2 existent. Si l'analyseur descendant est le seul analyseur envisagé REGLE1 et REGLE2 émettent des messages d'erreurs. Ces messages contiennent le numéro de la ligne contenant l'erreur (numéro de ligne du texte FACE) ; ce numéro est obtenu par REGLE1 et REGLE2 par recherche d'un élément FI. Si d'autres analyseurs sont disponibles les sous-programmes REGLE1 et REGLE2 se contentent de comptabiliser les infractions. Cette comptabilité est exploitée dans le module de commande de FADA afin d'orienter l'étude de la grammaire en vue d'une autre méthode d'analyse.

Dans tous les cas les sous-programmes REGLE1 et REGLE2 positionnent une variable NIVERR représentant le niveau d'erreur de la grammaire (niveau 3 pour une infraction à (D'1), niveau 4 pour une infraction à (D'2)).

Rappel des règles de déterminisme :

Soient $\alpha, \beta, \gamma \in V^*$, $A \in N$, $B, C \in T$

(D'1) $A ::= \alpha B \beta \mid \alpha C \gamma \Longrightarrow \mathcal{J}^*(B) \cap \mathcal{J}^*(C) = \emptyset$ et $B \overset{*}{\rightarrow} \Lambda \Rightarrow$ non $C \overset{*}{\rightarrow} \Lambda$

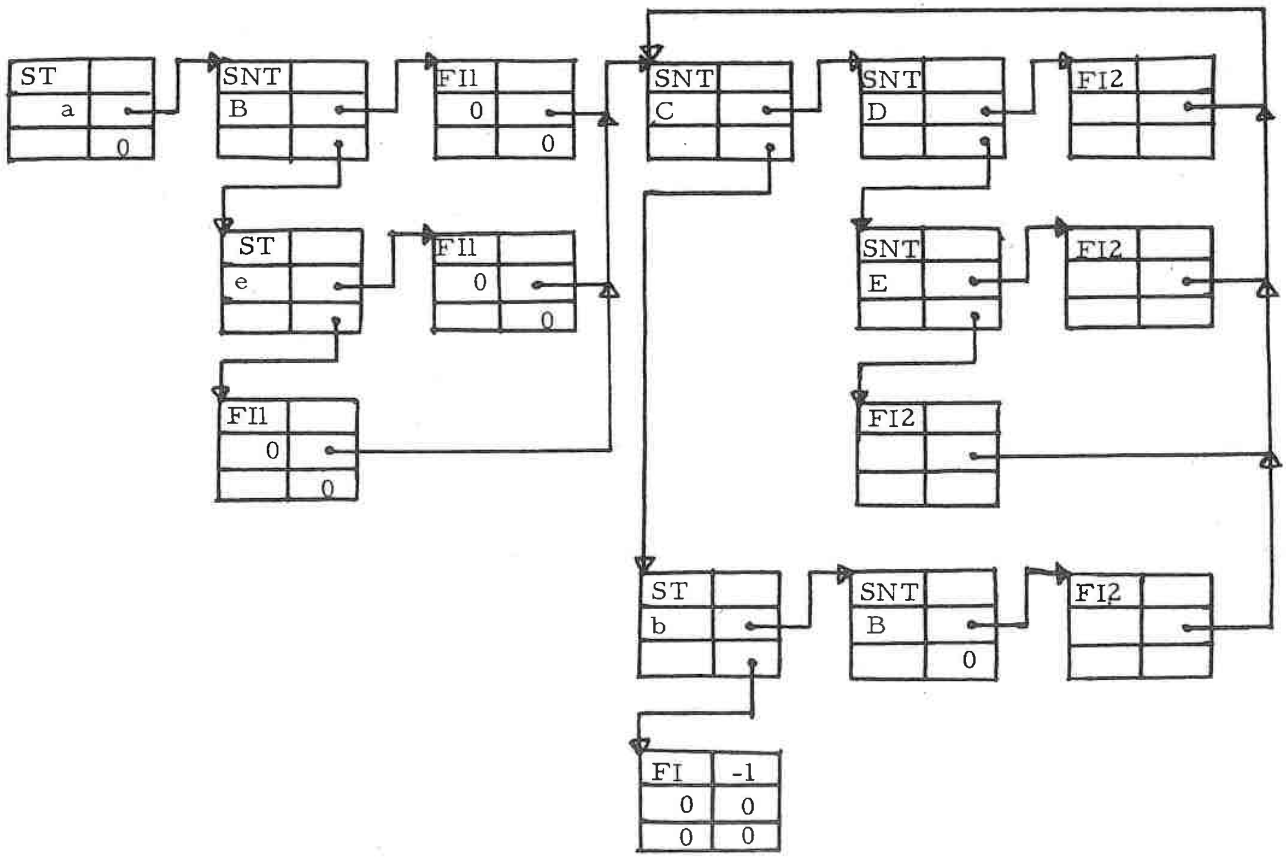
(D'2) $A ::= \alpha B \beta \mid \alpha \Longrightarrow \mathcal{J}^*(B) \cap vs(A) = \emptyset$ et non $B \overset{*}{\rightarrow} \Lambda$

Un contrôle supplémentaire est effectué sur les terminaux. VERIF s'assure que lorsqu'un choix peut se présenter entre plusieurs symboles terminaux, aucun n'est facteur gauche d'un autre. Ce contrôle se fait à l'aide du

sous-programme EMPILER qui reçoit un symbole en paramètre et empile toutes ses initiales terminales, en contrôlant pour chacune, qu'elle ne se trouve pas déjà empilée ou qu'elle n'est pas facteur gauche d'un symbole déjà empilé ou qu'un symbole déjà empilé n'en est pas facteur gauche. Le contrôle des règles vides ne pose pratiquement aucun problème. Lorsque EMPILER reçoit en paramètre un symbole B d'où dérive la chaîne vide : INT [B, LTDT] = 1. La valeur LTDT est donc empilée, comme s'il s'agissait du NDI d'une initiale terminale. La rencontre d'une alternative de B d'où dérive la chaîne vide provoquera une erreur puisque l'on tentera d'empiler deux fois le NDI LTDT. De même, lorsque dans une suite d'alternatives, une fin de règle est rencontrée, le NDI LTDT est empilé de manière à provoquer une erreur si dans la suite d'alternatives se trouvait un symbole d'où dérive la chaîne vide.

Le principe du sous-programme VERIF consiste à parcourir la TMS en examinant pour chaque non terminal la liste des premières alternatives, puis à appeler PARCOURS sur l'ensemble des règles de réécriture de chaque non terminal. A la rencontre de chaque élément ST ou SNT, PARCOURS appelle le sous-programme CONTROL qui examine la liste de ses successeurs.

Lorsque dans une suite d'alternatives, un élément de type FI1 ou FI2 est rencontré, ses successeurs doivent être considérés comme des alternatives supplémentaires à la liste de celles qui viennent d'être examinées. Exemple :



Le symbole terminal a peut dans les règles ci-dessus avoir comme successeurs : le non terminal B ou le terminal e ou le non terminal C ou le terminal b ou une fin de règle.

Le symbole non terminal C qui débute la partie récursive peut avoir comme successeurs : le non terminal D ou le non terminal E ou le non terminal C ou le terminal b ou une fin de règle.

o0o

Le contrôle des suites d'alternatives comportant un élément FI1 ne pose pas beaucoup de problèmes. Il en est autrement des suites d'alternatives comportant un élément FI2. En effet CONTROL ne peut accéder aux successeurs

des éléments FI2 car ceux-ci se trouvent sur le chemin arrière de PARCOURS et leurs liens de chaînage sont retournés.

Pour remédier à cet inconvénient, les éléments FI2 faisant partie d'une suite d'alternatives sont mis dans une liste chaînée (par la première moitié du 3ème mot de chaque élément). Ils reçoivent de plus dans le champ NDI, le NDI du membre de gauche et sont traités à la fin de VERIF.

Comme le sous-programme CONST de 2PASSAGE, le sous-programme CONTROL ne passe qu'une seule fois sur les parties répétitives des règles. Il utilise pour cela le même procédé que CONST. Avant l'appel de VERIF les marques introduites par 2PASSAGE sont retirées par un appel du sous-programme RAZ.

o0o

Finalement la partie centrale de VERIF s'écrit :

```
début aig Règle = (REGLE1, REGLE2) ;
     ETAP CONTROL ; TETE := MODIFIE := 0 ;
     aigi := 1  co positionnement de l'aiguillage pour que en cas d'erreur, le
                   sous-programme REGLE1 soit appelé co ;
     pour chaque NDI de symbole non terminal faire
     début 1ALT := AD1ELMT   := TMS [-NDI]  co paramètre de PARCOURS
                                                                  co ;
          P1ALT : TR:= TFRE [ 1ALT ]
                  si TR = FI alors début ⎧aigi := 2 ;
                                         ⎪pour chaque symbole terminal ST
                                         ⎪faire
                                TRAITFI  ⎨si VOISIN [NDI, ST]  alors
                                         ⎪ EMPILER (ST) ;
                                         ⎪EMPILER (LTDT) ; aigi := 1 ;
                                         ⎩allera SUITE
                                fin ;
                  si TR = FI1 alors début 1ALT:= SUC [ 1ALT ]  co les suc-
                                                                  cesseurs de l'élément FI1
```

```
                                        doivent être examinés comme
                                        des alternatives possibles co ;
                                        allera PlALT
                                fin ;
        co on est en présence d'un élément ST ou SNT co
        TR := NDI [lALT] ;
        EMPILER (TR) ;
        lALT := ALT [lALT] ;
        si lALT ≠ 0 alors allera PlALT ;
SUITE : co la liste des premières alternatives est examinée co
          remise à zéro de la pile construite par EMPILER ;
          PARCOURS
fin ;
si MODIFIE ≠ 0 alors TRFE [MODIFIE] := TYPE co remise en place du type
                        d'un élément modifié par CONTROL co ;
co il ne reste qu'à traiter la liste chaînée des éléments FI2 construite par
CONTROL. Le pointeur de tête de liste est TETE co
LISTFI2 : si TETE ≠ 0 alors
              début R := TETE ;
                    NDI := NDI [R]  co NDI = NDI du membre de gauche co ;
                    NDI [R] := 0  co remise à zéro du champ NDI co ;
                    TETE := pointeur de chaînage de l'élément FI2 ;
                    remise à zéro du lien de chaînage de l'élément FI2 ;
                    ETAP RETABLIR ;
                    ADlELMT := TMS [-NDI];
                    PARCOURS co le sous-programme RETABLIR place
                    dans R14 l'adresse de la liste d'alternatives contenant
                    l'élément FI2 co ;
              ALT :  TR := TFRE [R14] ;
                    si TR = FI2 alors début R14 := SUC [R14] ; allera ALT fin ;
                    si TR = ST ou TR = SNT alors début EMPILER (R14) ;
                                                       R14 := ALT [R14] ;
```

```
                                        allera ALT
                                    fin
                        co on est en présence d'un élément de type FI co
                        aigi := 2 ;
                        EMPILER (LTDT) ;
                        remise à zéro de la pile construite par EMPILER ;
                        allera LISTFI2
                    fin
        fin
```

Le sous-programme CONTROL reçoit en paramètre dans R6, l'adresse d'un élément ST, SNT ou FIl. Il examine la liste des successeurs de cet élément.

```
procédure CONTROL ;
début si MODIFIE ≠ 0 alors début TFRE [MODIFIE] := TYPE ;
                                  MODIFIE := 0
                             fin ;
      TR := TFRE [R6] ;
      si TR = FIl alors Return co pour un élément FIl il n'y a rien à faire co ;
      lALT := SUC [R6]  co lALT = ler successeur co ; lRS := lALT;
CTRL : TR := TFRE [lALT] ;
        si TR = FI alors début TRAITFI ;
                               Return
                          fin ;
        si TR = FIl alors début lALT := SUC [lALT] ; R := lALT;
                          ALTFIl : TR := TFRE [lALT] ;
                                   si TR ≠ FI alors début TR := NDI [lALT];
                                                          EMPILER (TR) ;
                                                  lALT := ALT [lALT] ;
                                                  allera ALTFIl
                                              fin ;
                          TRAITFI ;
                          si l'élément d'adresse relative R est marqué alors
```

```
                        début TYPE := TFRE [R] ;
                              TFRE [ MODIFIE ] := FI ;
                              MODIFIE := R ;
                        fin
                        sinon marquer l'élément R ;
                  remise à zéro de la pile construite par EMPILER ;
                        Return
               fin ;

   si TR = FI2 alors début si lALT ≠ lRS alors co l'élément FI2 apparaît
                              après une suite d'alternatives. Il doit être placé
                              en liste co
                        début 5ième demi mot de l'élément lALT := TETE ;
                              NDI [lALT] := NDI ;
                              TETE := lALT ;
                        fin ;
                              remise à zéro de la pile construite par EMPI-
                                                      LER ; Return
                        fin ;
   co on est en présence d'un élément ST ou SNT co
   TR := NDI [lALT] ;
   EMPILER (TR) ;
   lALT := ALT [lALT] ;
   si lALT ≠ 0 alors allera CTRL ;
   remise à zéro de la pile construite par EMPILER ; Return
fin
```

4.10. CONTROLE DES TERMINAUX : SOUS PROGRAMME EMPILER

Le sous-programme EMPILER est chargé de vérifier la condition de déterminisme suivante :

Si un choix se présente entre plusieurs symboles terminaux, aucun ne doit être facteur gauche de l'autre.

EMPILER reçoit en paramètre des NDI de symboles terminaux ou non terminaux. L'ensemble des NDI des initiales terminales ($\mathcal{J}^*$) de chaque symbole reçu en paramètre est mémorisé après contrôle avec ceux déjà mémorisés. EMPILER doit donc comparer deux à deux des symboles terminaux et décider si l'un d'eux est facteur gauche de l'autre.

Trois cas peuvent se présenter :

1) comparer entre eux deux terminaux du type "chaîne de caractères".

2) Comparer un terminal "chaîne de caractères" et un terminal "automate".

3) Comparer entre eux deux terminaux "automates".

Le premier cas ne présente pas de difficulté.

Examinons le deuxième cas. Il s'agit de comparer un terminal "chaîne" t et un terminal "automate" A.

Soit L (A) le langage accepté par A.

si $\exists \alpha, \beta$ tels que $\alpha \in$ L (A) et t = $\alpha \beta$ on dira que le terminal A est facteur gauche du terminal t.

si $\exists \gamma, \delta$ tels que $\gamma \in$ L (A) et γ = t δ on dira que t est facteur gauche de A.

Il est possible d'avoir à la fois t facteur gauche de A et A facteur gauche de t, mais cela n'a guère d'importance ici. Ce qui importe est de savoir si oui ou non l'un est facteur gauche de l'autre.

Les deux éventualités peuvent être détectées en tentant de reconnaître la chaîne t à l'aide de l'automate A de la manière suivante :

```
début ETAT := état initial de A ;
      i := 0;
iter : ETAT := AUTOMATE [ETAT, caractère i de t] ;
      si ETAT = 0 alors les deux terminaux sont compatibles, aucun n'est
                        facteur gauche de l'autre ;
      si ETAT est un état final alors A est facteur gauche de t ;
```

i := i + 1 ;

si caractère i de t = 0 (marque de fin de t) alors t est facteur gauche de A;

allera iter

fin

La troisième possibilité consiste à comparer entre eux deux terminaux "automates" A1 et A2

si $\exists \alpha, \beta, \gamma$ tels que $\alpha \in L(A2)$, $\beta \in L(A1)$ et $\alpha = \beta\gamma$ alors A1 est facteur gauche de A2.

Il est bien sûr possible d'avoir en même temps A1 facteur gauche de A2 et A2 facteur gauche de A1.

Le premier test à faire consiste à s'assurer que A1 et A2 sont bien des automates différents. En effet un même automate peut figurer dans plusieurs terminaux différents s'il y est associé à des procédures lexicographiques différentes. Dans le cas où les états initiaux des deux automates sont identiques, on dira que chacun est facteur gauche de l'autre (en fait L (A1) = L (A2)).

S'il n'en n'est pas ainsi, il faut chercher les facteurs gauches communs aux deux langages.

Pour cela, partant des états initiaux de A1 et A2, l'algorithme sélectionne les chemins des deux automates contenant des arcs portant des valeurs communes (mêmes caractères reconnus). Si l'on aboutit ainsi à un état final de l'un des deux automates, cela signifie que ce dernier est facteur gauche de l'autre. L'algorithme s'arrête soit lorsqu'il atteint un état final de l'un des automates, soit lorsqu'il n'y a plus d'arcs portant des valeurs identiques.

L'algorithme doit tenir compte de l'existence possible de circuits qui pourraient l'entraîner à boucler indéfiniment.

Soit α un facteur gauche commun à L (A1) et L (A2). Après la reconnaissance de α, A1 se trouve dans l'état s1 et A2 dans l'état s2. Supposons qu'il existe un caractère a tel que AUTOMATE [s1, a] = s'1 et AUTOMATE [s2, a] = s'2 et que les états s'1 et s'2 aient déjà été rencontrés au cours de l'analyse

de α. Il est alors inutile de poursuivre l'étude d'un facteur gauche commun à partir de s'1 et s'2. En effet une telle poursuite peut éventuellement conduire à une boucle infinie mais de toutes façons il faut poursuivre l'étude d'un facteur gauche commun à partir des états déjà rencontrés en cherchant une autre voie.

Pratiquement l'algorithme de reconnaissance des facteurs gauches est un algorithme de recherche de chemins dans le graphe G = (E, Γ')

où E = Q x R Q ensemble des états de A1

R ensemble des états de A2.

$(q_1, r_1)\ \Gamma\ (q_2, r_2)$ $\Longleftrightarrow$ il existe un calcul de A1 et un calcul de A2 acceptant le même mot α se terminant par q_1 et r_1 et $\exists$ a tel que (q_1, a, q_2) soit une transition de A1 et (r_1, a, r_2) soit une transition de A2.

L'algorithme utilise une pile : CHEMIN dont chaque élément occupe 3 octets :

1er octet = état de A1

2ème octet = état de A2

3ème octet = caractère du facteur gauche commun

Finalement l'algorithme s'écrit :

```
début ETA1 := état initial de A1 ;
      ETA2 := état initial de A2 ;
      h := 0  co h est la hauteur de la pile ; h := 0 indique une pile vide co ;
examin : i := X'39'  co indice du caractère à examiner co;
carsvt : i := i + 1 ;
         si  i > X 'FA' alors allera dépiler ;
         ST1 := AUTOMATE [ETAT1, i] ;
         ST2 := AUTOMATE [ETAT2, i] ;
         si ST1 = 0 ou ST2 = 0 alors allera carsvt ;
         si AUTOMATE [ST1, X'FB'] ou AUTOMATE [ST2, X'FB'] alors
             l'algorithme s'arrête, l'un des automates étant facteur gauche de
             l'autre ;
         pour hh := 1 pas 1 jusqua h faire co recherche des circuits co
```

```
        si CHEMIN [hh,1]  = ST1 alors pour hh := 1 pas 1 jusqua h faire
                                        si CHEMIN [hh,2]  = ST2 alors
                                            allera carsvt ;
        CHEMIN [h]  := (ETAT1, ETAT2, i) ;
        ETAT1 := ST1 ;
        ETAT2 := ST2 ;
        allera examin ;
depiler : si h = 0 alors fin de l'algorithme, aucun des automates n'étant fac-
                                teur gauche de l'autre ;
        (ETAT1, ETAT2,i) := CHEMIN [h]  ;
        h := h - 1 ;
        allera carsvt
fin
```

Dans cet algorithme, la hauteur de la pile CHEMIN est bornée à 252. En effet, soient nl le nombre d'états de l'automate A1 et n2 le nombre d'états de l'automate A2.

Le nombre d'états de A1 pouvant se trouver empilé à un instant donné est au plus égal à nl - 1 (un état final n'est jamais empilé et A1 comporte au moins un état final).

De même le nombre d'états de A2 pouvant se trouver empilé à un instant donné est au plus égal à n2 - 1.

Chaque élément de CHEMIN contient un couple d'états constitué d'un état de A1 et d'un état de A2. Pour qu'un tel couple soit empilé il faut que l'un au moins des deux états ne figure pas encore dans CHEMIN. On peut imaginer que lorsqu'un état est empilé dans CHEMIN, cet état est marqué (cela peut d'ailleurs être une méthode de réalisation de l'algorithme évitant la recherche dans CHEMIN à chaque empilement). Pour qu'un empilement se fasse il faut que l'un au moins des deux états ne soit pas marqué. Chaque empilement se traduit par le marquage d'au moins un état non encore marqué. La hauteur de la pile est donc au plus égal au nombre d'états marqués moins un (car le

premier empilement provoque le marquage des deux états initiaux). Le nombre d'états marqués est borné par (n1 - 1) + (n2 - 1). La hauteur de la pile est donc au plus n1 + n2 - 3, n1 + n2 est au plus égal au nombre d'états possibles pour tous les automates d'une grammaire, soit 255.

La hauteur de la pile ne dépassera jamais 252 groupes de 3 octets soit 189 mots.

Le tableau CHEMIN peut donc être réservé de façon fixe dans le programme.

4.11 CONTROLE DES CARACTERES A SAUTER : T E S T S A U

Le rôle de ce sous-programme est simplement de vérifier que les caractères à sauter ne figurent pas à l'intérieur d'un terminal de la grammaire.

Pour chaque caractère à sauter de la grammaire, autre que X'00' (saut de ligne) TESTSAU effectue deux contrôles :

1) que le caractère ne figure à l'intérieur d'aucun terminal "chaîne" de la table des terminaux.

2) Que la colonne de la matrice AUTOMATE correspondant à ce caractère ne contient que des zéros.

Ces vérifications sont aussi valables pour l'analyse descendante que pour l'analyse ascendante. Les anomalies rencontrées conduisent donc à la sortie de messages d'erreurs et à l'affectation d'un niveau d'erreur élevé à la grammaire.

4.12 LE SOUS PROGRAMME : FINAL

FINAL restitue la mémoire obtenue dynamiquement, place le vecteur de renseignements du tableau INT dans les 4 mots réservés à cet effet dans le vecteur de renseignements et place dans le mot zéro du vecteur de renseignements la valeur zéro, indiquant que la grammaire est destinée à une analyse descendante.

4.13 LE SOUS PROGRAMME : ECRIRE

ECRIRE place la grammaire sur disque où elle constitue un membre d'un fichier partitionné. Ce membre comporte :

- le vecteur de renseignements
- la table des caractères à sauter
- la matrice AUTOMATE
- la table des terminaux
- la table des non terminaux
- le GRG
- la matrice INT

Ce membre est constitué d'enregistrements de longueur variable.

CHAPITRE V

LE SOUS-PROGRAMME D'ANALYSE DESCENDANTE

A N A L D E S

L'ANALYSEUR DESCENDANT ANALDES

5.1 LES DONNEES DE L'ANALYSEUR

L'analyseur descendant a comme données la grammaire traitée par FADA d'une part le texte source à analyser d'autre part. Au moment où l'analyseur proprement dit est activé, la grammaire se trouve déjà en mémoire. Elle y a été placée par le module PREAMBULE commun à l'analyseur ascendant et descendant, chargé de choisir la grammaire pertinente dans un fichier pouvant en contenir plusieurs, ainsi que de l'analyse de la carte de commande définissant les paramètres de compilation (listing, programme objet ...).

Le module PREAMBULE détermine le type d'analyseur correspondant à la grammaire et l'appelle en lui communiquant :

- l'adresse en mémoire de la grammaire (dans le mot T)
- le numéro de règle ayant l'axiome pour membre de gauche (dans le mot NRAX)

il positionne un aiguillage dans le sous-programme de sortie d'un numéro de règle selon que l'appel en cours est un appel d'ANALYSE ou d'ANAPPLYSE.

L'analyseur descendant commence par une série d'initialisations comportant en particulier la recherche du NDI (numéro d'identification) de l'axiome.

Le texte source est lu au moyen du module LECT décrit plus loin.

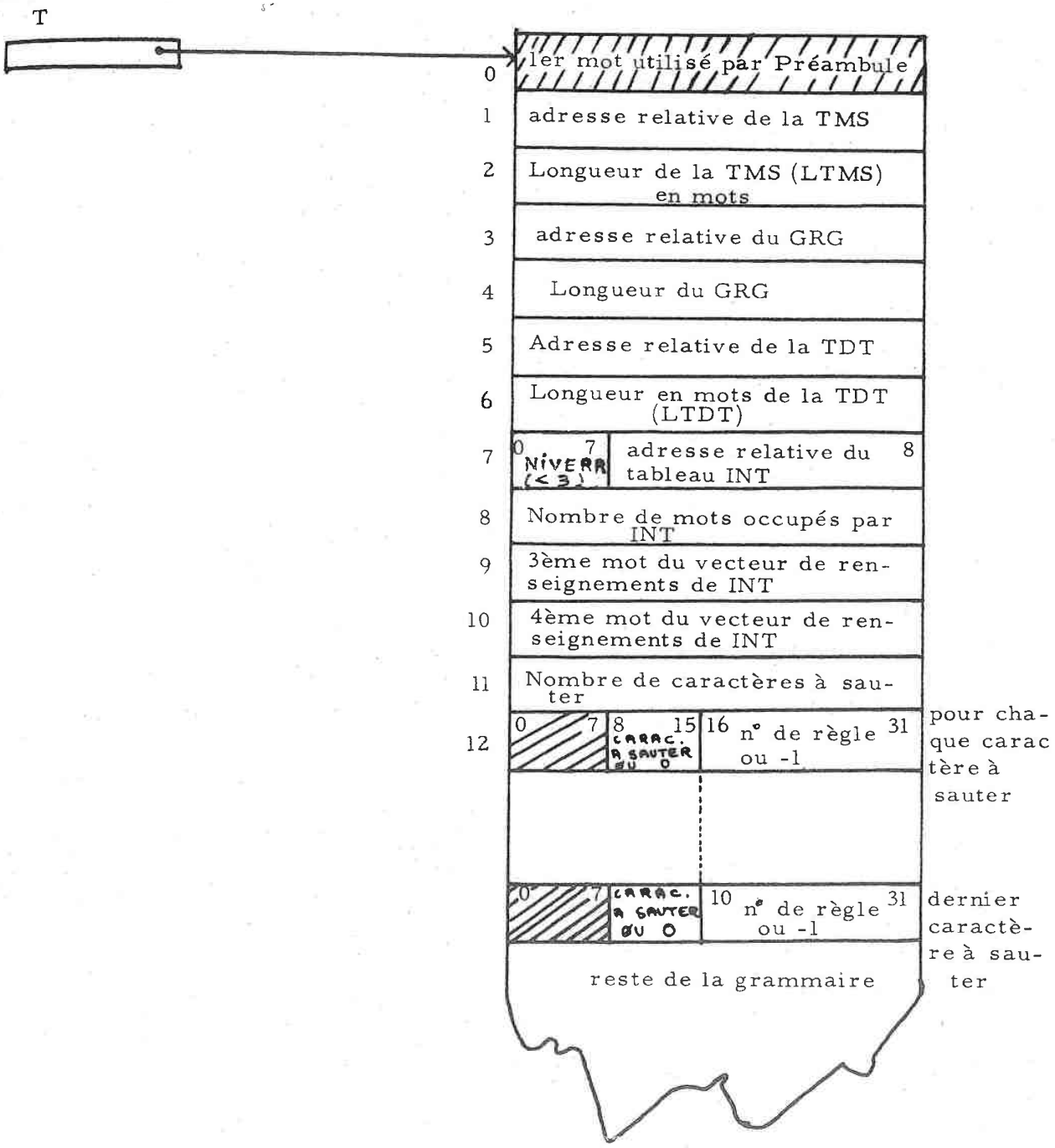
Rappel du format de la grammaire

La grammaire comporte un premier mot utilisé par PREAMBULE inutile pour l'analyseur.

On trouve ensuite un vecteur de renseignements, puis les tables décrites dans ce vecteur de renseignements.

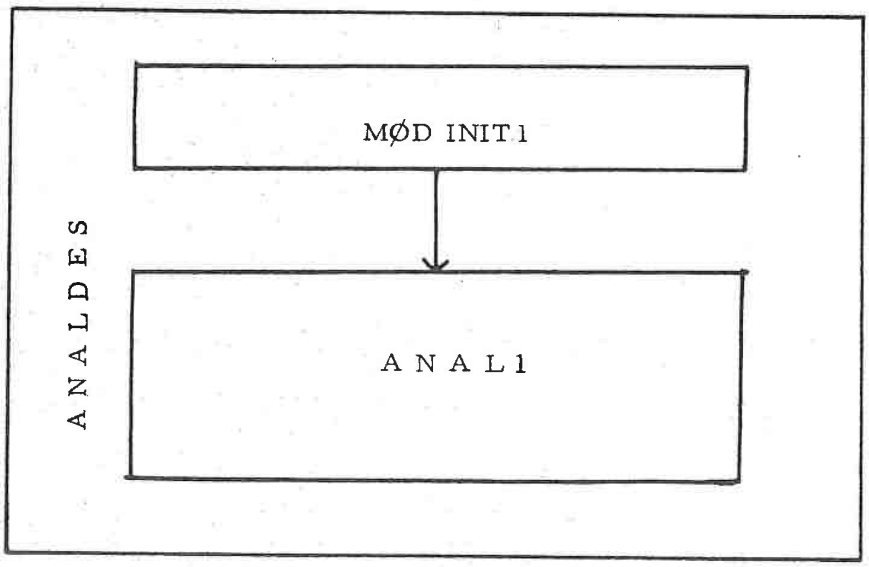
Le vecteur de renseignements contient :

- l'adresse relative puis la longueur en mots de la TMS (table des non terminaux),
- l'adresse relative puis la longueur en mots du GRG (graphe représentant la grammaire),
- l'adresse relative puis la longueur en mots de la TDT (table des terminaux),
- sur 4 mots la description du tableau INT des initiales terminales,
- le nombre de caractères à sauter,
- les caractères à sauter ainsi que le numéro de règle qui les accompagne.



Le module MODINIT1 est suivi de l'analyseur descendant proprement dit qui va maintenant être décrit au sein du module ANAL1.

Le plan de l'analyseur descendant est donc le suivant :



5.2. LE MODULE D'INITIALISATIONS MODINIT1

Commence par initialiser certaines variables utilisées dans le reste de l'analyseur en particulier :

LTMS := longueur de la TMS

LTDT := longueur de la TDT

NSAU := nombre de caractères à sauter

SL := 1 si parmi les caractères à sauter il y a le caractère X'00' (accompagné d'un numéro de règle ≠ -1) (saut de ligne), 0 sinon.

NORSL := si SL = 1 alors le numéro de règle à sortir en cas de saut de ligne.

o0o

Il est ensuite nécessaire de déterminer NDIAX, le NDI de l'axiome de la grammaire. Il faut pour cela trouver la règle l'ayant pour membre de gauche au moyen de NRAX (numéro de règle de l'axiome).

Cela se fait en parcourant la grammaire au moyen du sous-programme PARCOURS utilisé aussi dans FADA :

```
début ETAP CHERCH1, CHERCH2 ; (préparation de parcours)
      trouvé := 0 ; co indique la rencontre de la règle cherchée co
      pour i := -1 pas 1 jusqua-LTMS faire
      début co positionnement de paramètre de PARCOURS : ADIELTMT co
            PARCOURS (TMS [i] ) ;
            si trouvé ≠ 0 alors début
                                    NDIAX := i ;
                                    allera finch
                              fin
      fin ;
      si trouvé = 0 alors début imprimer le message :
                                    "grammaire sans axiome" ;
                                    abandon du travail
                        fin ;
finch : fin
```

Les sous-programmes CHERCH1 et CHERCH2

A l'appel de ces sous-programmes par PARCOURS, RPX2 contient l'adresse relative d'un élément qui peut être un élément FI1 dans le cas de CHERCH1 qui est un élément FI ou FI2 dans le cas de CHERCH2.

Les deux sous-programmes peuvent avoir une partie commune

```
CHERCH1 : début R1 := TFRE [RPX2]
                si R1 = FI1 alors com : début  R1 := IDENT [RPX2] ;
                                        si R1 = NRAX alors
```

```
                                    début trouvé := 1 ;
                                         ETAP
                                    fin
                               fin
          fin
CHERCH2 : allera com
```

5.3 PRINCIPE DE L'ALGORITHME D'ANALYSE

L'analyseur descendant est déterministe et utilise le principe de la lecture d'un caractère (ou plutôt ici d'un terminal) à l'avance.

Au début de l'analyse on part de l'axiome et on tente l'analyse en utilisant l'une de ses règles de réécriture. Le choix s'effectue au niveau de chaque alternative grace au tableau des initiales et au terminal lu en avance. Lorsqu'un symbole est choisi, il est analysé complétement selon le même processus que l'axiome avant de poursuivre dans la règle. Il est alors nécessaire d'utiliser une pile pour mémoriser l'ensemble des règles en cours d'analyse. Chaque élément de la pile sera un mot et contiendra l'adresse relative dans le GRG de l'endroit où poursuivre l'analyse de la règle. Lorsqu'une règle est terminée la pile diminue.

La pile d'analyse est réservée de façon statique. On ne peut modifier sa taille qu'en réassemblant l'analyseur.

Le fait que l'analyse se fasse sur des terminaux et non sur de simples caractères présente des difficultés au niveau de la lecture à l'avance d'un terminal.

En effet il est possible qu'un terminal ne puisse pas être lu de manière déterministe (un terminal peut être facteur gauche d'un autre). On ne peut être sûr de choisir un terminal de manière déterministe que parmi les initiales terminales d'un symbole (cf. chapitre II).

Pour permettre cela on pratique ce que l'on pourrait appeler une lecture à l'avance différée.

Lorsque l'on avance dans une règle on effectue une lecture à l'avance non pas dans l'absolu, mais en fonction du symbole à analyser. Une telle lecture est possible ou non ; si elle ne l'est pas on choisira une autre alternative (s'il en existe une).

Lorsqu'on analyse un nouveau symbole il faut tester un indicateur pour savoir si un terminal à l'avance a été lu ou non.

La lecture d'un terminal se fait toujours au moyen du sous-programme LIRE qui reçoit en paramètre le NDI d'un symbole ; il essaie de lire dans le texte source une initiale terminale de ce symbole. LIRE a pour résultat une variable qui peut avoir la valeur succès (si un terminal a pu être lu) ou échec sinon. Dans le cas de succès il fournit en plus dans c le NDI du terminal lu.

Lorsque l'examen d'une règle passe sur un élément FI, FI1 ou FI2 contenant un numéro de règle différent de -1, l'analyseur appelle le sous-programme SORTIR avec ce numéro pour fournir la chaîne post fixée résultat de l'analyse. Le module SORTIR est commun avec l'analyseur ascendant.

5.4. ALGORITHME D'ANALYSE

```
début R := TMS [NDIAX] ; empiler (R) co sur la pile d'analyse co ;
      LIRE (c en fonction de NDIAX co NDIAX = NDI de l'axiome co)
      si échec alors ERREURD ; lec := faux ;

      co la procédure empiler doit prévoir un test de débordement de pile.
      co
début : T := TFRE [R]; SASAUTER ;
        co T peut recevoir l'une des valeurs SNT, ST, FI1, FI2 ou FI.
```

Chaque type a son traitement propre. On a donc maintenant un aiguillage qui choisit le bon traitement selon la valeur de T.co

```
si T = SNT alors allera TSNT ;
si T = ST  alors allera TST ;
si T = FI1 ou FI2 alors allera TFIi ;
si T = FI alors allera TFI ;
TSNT : co traitement d'un élément SNT co
    T := NDI [R];
    si lec = vrai alors début LIRE (c en fonction de T) ;
                          si échec alors allera choix ;
                          lec := faux ;
                          allera empiler
                      fin ;
    si INT [T, c] alors empil : début R := IDENT [R] ;
                                empiler (R) ;
                                allera début
                            fin ;
    allera choix ;
ST : co traitement d'un élément ST co
    T := NDI [R], SASAUTER ;
    si lec = vrai alors début LECT (T) ;
                      co le sous-programme LECT est utilisé à la place
                      de LIRE dans le cas d'un terminal co
                      si échec alors allera choix ;
                      allera suite
                  fin ;
    si T = c alors suite : début R:= SUC [R] ;
                           Remplacer le sommet de la pile d'analyse
                           par R ;
                           lec := vrai co dans l'analyse avec lecture à
                           l'avance, c'est maintenant qu'il faut lire un
```

terminal ; ici cette lecture est différée, on note dans lec qu'elle n'est pas encore faite co, allera début

fin ;

choix : co Cette partie du programme est commune à SNT et ST

On s'est aperçu que l'élément du GRG dont l'adresse est dans R ne permet pas de filtrer les textes source. On va alors essayer de choisir une autre alternative. S'il n'y en a pas on se demande si l'élément dont l'adresse est dans R permet de filtrer la chaîne vide ; si oui l'analyse peut se poursuivre normalement ; sinon il y a une erreur co

X := ALT (R);

si X = 0 alors co Il n'y a pas d'autre alternative possible co

début si T < 0 alors début si INT [T, LTMS]

alors allera empil

fin ;

ERREURD

fin ;

R := X ; co R := adresse de la nouvelle alternative co

Remplacer le sommet de la pile par R ;

allera début ;

TFIi : co traitement d'un élément FI1 ou FI2 co

T := IDENT [R];

si T ≠ -1 alors sortir (T) ;

R := SUC [R] ;

Remplacer le sommet de la pile par R ;

allera début ;

TFI : co traitement d'un élément FI co

T := IDENT [R];

si T ≠ - 1 alors sortir (T) ;

si pile d'analyse vide alors allera fin ;

```
        Dépiler ;
        R := nouveau sommet de la pile d'analyse;
        R := SUC [R] ;
        Remplacer le sommet de la pile par R ;
        allera début ;
fin :   co traitement de fin d'analyse co
        si (si B = 1 alors BUF1 sinon si B=2 alors BUF2 sinon RESERVE)
                                    [I]≠ EØD
        alors imprimer un message "fin de fichier non attendu" ;
finl : sortir (marque de fin de chaîne soit -1) ;
        libération de mémoire ;
        Retour ;
fin
```

5.5. LE SOUS-PROGRAMME L I R E (c EN FONCTION DE NDI)

Ce sous-programme tente de lire dans la chaîne d'entrée un terminal de NDI : c qui soit une initiale du symbole dont le NDI est passé en paramètre.

Le sous-programme LIRE utilise un autre sous-programme : LECT (commun à l'analyseur ascendant et descendant). LECT tente de reconnaître dans le texte source un terminal dont le NDI lui est donné en paramètre. Il répond en positionnant une variable à succès ou échec.

Le rôle de LIRE est alors d'appeler LECT pour chaque terminal initial de son paramètre et de retenir éventuellement dans c le numéro de celui qui conduit LECT à succès.

LIRE doit être appelé uniquement pour un non terminal (NDI < 0) ; pour faire une lecture d'un terminal il faut utiliser directement LECT.

```
début pour i := 0 pas 1 jusqua LTDT -1 faire
      début si INT [NDI, i]  = 1 alors début LECT (i) ;
                                         si succès alors début c := i ;
                                                         allera retour
                                                     fin
                                   fin
      fin;
retour : fin
```

On peut augmenter l'efficacité de ce programme en considérant que l'on suit dans INT une ligne. Les valeurs s'obtiennent par décalages successifs.

5.6. LE SOUS-PROGRAMME SASAUTER

Lorsque le sous-programme LECT rencontre des caractères à sauter dans le texte source, il ne produit pas les numéros de règles correspondants (figurant dans la table TCS de la grammaire) car ces numéros apparaîtraient dans la chaîne postfixée au moment de la lecture à l'avance du terminal et non, comme on s'y attend logiquement, à la suite du terminal qui les contient. Les numéros de règle correspondant sont empilés par LECT dans la table PILSAU.

Le sous-programme SASAUTER est chargé, chaque fois qu'il est appelé, de sortir les numéros de règle accumulés dans PILSAU et de vider PILSAU.

Le sous-programme d'analyse appelle SASAUTER avant de commencer l'analyse d'un nouveau terminal ou non terminal (cf. 5.4).

5.7. LE SOUS PROGRAMME ERREURD

Le sous programme ERREUD est appelé par l'analyseur descendant au moment de la détection d'une erreur.

Le rôle de ERREURD est :

- de commander l'émission d'un message d'erreur spécifique (au moyen du sous-programme MESSERR) ; pour cela il faut rechercher le numéro de l'erreur.

- de tenter de repartir dans l'analyse, dans la mesure où le texte soumis n'est pas encore complètement analysé. Bien entendu, dès qu'une erreur est détectée le résultat de l'analyse dans la chaîne postfixée n'a plus une bonne signification, mais il faut néanmoins poursuivre l'analyse de manière à tenter de détecter toutes les erreurs possibles du texte source.

Le programme ERREURD n'a pas à comptabiliser les erreurs, ceci est fait par le sous-programme MESSERR chargé de gérer les messages. d'erreurs.

Le sous-programme ERREURD comportera donc deux parties bien distinctes:

La première recherche le numéro de l'erreur, c'est-à-dire le numéro de la règle en cours d'analyse ; pour cela il faut trouver un élément FI, FI1 ou FI2 au moyen du sous-programme PARCOURS. Cet élément doit contenir effectivement un numéro de règle et non pas -1. Cette première partie se termine par l'appel de MESSERR.

La deuxième tente de poursuivre l'analyse (dans la mesure où la fin de fichier n'est pas rencontrée). Pour cela deux actions sont combinées :

dépilement de la pile d'analyse de manière à reprendre l'analyse à un niveau supérieur

et élimination de caractères dans le texte source.

Cet algorithme de reprise repose sur le principe suivant :

On suppose d'abord que l'erreur correspond à l'émission d'un certain nombre de caractères. On recherche alors un élément R de la pile d'analyse tel que R $\mathcal{G}\mathcal{J}^*$ c, c étant le terminal qui se présente dans le texte source, de

manière à sauter la partie manquante de la ramification. Si cette reprise est impossible (aucun élément de la pile ne satisfaisant à la condition R $\mathcal{S}$ $\mathcal{Y}^*$ c), on considère qu'une partie du texte source n'est pas analysable. La pile est restaurée dans l'état où elle se trouvait au moment de la détection de l'erreur, et des caractères du texte source sont sautés, jusqu'à ce que l'on puisse mettre en évidence un terminal c vérifiant avec un élément R de la pile la relation R $\mathcal{S}$ $\mathcal{Y}^*$ c.

1) Recherche du numéro d'erreur

```
début  ETAP  AFFECNO, AFFECNO ; NO := -1
       pour chaque élément S de la pile d'analyse dans l'ordre du sommet
       vers la base faire
       début  PARCOURS (S) ;
              si NO ≠ - 1 alors allera trouvé ;
       fin ;
trouvé :  MESSERR (NO)
fin.
```

Le sous-programme AFFECNO s'écrit :

```
début  co RPX2= adresse de l'élément examiné par PARCOURS co ;
       R:= LTFRE [ RPX2 ]
       si R = ST ou SNT alors retour ;
       R:= IDENT [ RPX2 ] ;
       si R ≠ -1 alors début NO:= R ;  ETAP  fin ;
       retour
fin
```

2) Recherche d'un point de reprise dans l'analyse

Soit h la hauteur de la pile d'analyse au moment de l'appel de ERREURD.

si (si B = 1 alors BUF1 sinon si B = 2 alors BUF2 sinon RESERVE)
[I] = EØD
alors allera finl co (cf. annexe 2) il faut adresser le buffer de lecture en cours d'utilisation co ;

hh := h ;

succ : R:= pile [h] ; h:= h-1 ;

R:= SUC [R] ;

tsucc : T:= TFRE [R] ;

si T = FI alors allera descendre ;

si T = FI1 ou FI2 alors allera succ ;

NDIES:= NDI [R] ; co NDIES = NDI d'essai de reprise co

si lec = vrai alors début si NDIES < 0 alors LIRE (c en fonction de NDIES)
sinon début c := NDIES ;
LECT (c) fin ;
si succès alors début lec := faux ;
allera début
fin
sinon allera choixerr
fin ;

si NDIES < 0 alors début si NDIES = c alors allera début fin
sinon début si INT [NDIES, c] alors allera début fin ;

choixerr : R:= ALT (R) ;

si R ≠ 0 alors allera tsucc ;

descendre : co on vient de constater que le caractère se présentant dans le texte source ne peut être analysé par l'élément suivant de la règle en cours d'analyse. On reprend une règle précédente empilée sur la pile d'analyse co

```
        si h ≠ 0 alors allera succ ;
co cas où toute la pile a été descendue sans pouvoir reprendre l'analyse. On
    remet alors la pile dans son état initial et on avance dans le texte source
    pour recommencer la même opération co
        h := hh ;
        lec := faux ;
suivre: SASAUTER ; AVANCCAR ;
        pour c := 0 pas 1 jusqua LTDT -1 faire
        début LECT (c) ; si succès alors allera succ fin ;
        si (si B = 1 alors BUF1
                    sinon si B = 2 alors BUF2
                                sinon RESERVE) [ I ] = EØD
          alors allera finl ;
        allera suivre ;
```

o0o

On trouvera en annexe 2 l'étude des sous-programmes :

LECT (lecture d'un terminal)
MESSERR (émission d'un message d'erreur)
AVANCCAR (avancement d'un caractère dans le texte source).

Ces sous-programmes de service sont communs à tous les analyseurs.

C H A P I T R E VI

Etude de la grammaire

Adéquation d'une analyse ascendante

6.1. PRINCIPAUX ELEMENTS D'UNE GRAMMAIRE DESTINEE A L'ANALYSE ASCENDANTE.

Lorsque FADA constate, après exécution du sous-programme VERIF, que la grammaire étudiée ne peut faire l'objet d'une analyse descendante, il doit étudier la possibilité d'une analyse ascendante; et si celle-ci est possible, transformer la grammaire de manière à obtenir une nouvelle représentation adaptée à ce type d'analyse. Cette nouvelle représentation correspond à la structure logique décrite au chapitre 2 pour l'analyse ascendante. L'accès aux règles se fait par le dernier symbole du membre de droite. Les fins de règles identiques sont partagées. Les règles se terminent par un terminateur, dit élément **FI**, donnant le NDI du membre de gauche et le numéro de règle. Cette représentation de la grammaire s'accompagne des matrices de précédence.

Avant d'étudier les algorithmes permettant de passer d'une représentation à l'autre, voyons le format physique de cette nouvelle représentation.

La grammaire elle-même sera constituée de 3 tables. Il devra en outre y être adjoint la matrice de précédence. Ces tables sont :

- la table ENTREE

Cette table contient un mot pour chaque symbole (terminal ou non terminal). Chaque mot de cette table donne accès aux règles dont ce symbole est une finale. Si un symbole n'est la finale d'aucune règle, l'élément correspondant de la table ENTREE aura la valeur zéro.

La table ENTREE contient une entrée spéciale (la dernière) pour les règles vides. La dimension de la table ENTREE est donc :

ENTREE[-LTMS:LTDT] (avec un mot par élément)

- la table des terminaux (TDT)

Table strictement identique à celle de la représentation adaptée à l'analyseur descendant. Cette identité de représentation des terminaux autorise les deux analyseurs à partager le même sous-programme LECT de lecture d'un terminal, ainsi que le sous-programme

MESSERR gérant les messages d'erreur.

- le graphe représentant les règles (TGA)

Cette table contient la représentation de toutes les règles de la grammaire sous une forme adaptée à l'analyse ascendante. Cette forme correspond à la structure logique définie au chapitre 2.

C'est à l'intérieur de cette table que pointe chaque mot de la table ENTREE.

Ce graphe est en fait constitué des réfléchis des membres de droite des règles, le dernier symbole de la règle étant sous-entendu puisque c'est lui qui détermine le point d'entrée. Chaque branche du graphe se termine par un élément spécial (élément FI) indiquant le NDI du membre de gauche ainsi que le numéro de règle.

Chaque symbole d'une règle est représenté dans ce graphe par un élément de 1 mot.

Les éléments d'une même règle figurent les uns à la suite des autres (le lien de succession est représenté par la fonction successeur en mémoire).

Les fins de règles identiques sont systématiquement mises en facteur. Les alternatives possibles à un symbole sont notées dans l'élément qui le représente au moyen du lien "alt".

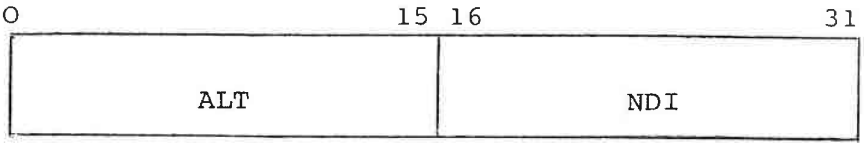
La fin d'une règle est toujours marquée par un élément FI qui n'a bien sûr aucun successeur mais peut avoir une alternative.

Chaque élément FI contient les informations suivantes :

- NDI du membre de gauche (MG)
- numéro de règle
- alternative éventuelle.

Ces 3 informations ne peuvent tenir sur un mot . En conséquence, chaque élément FI sera formé de deux mots.

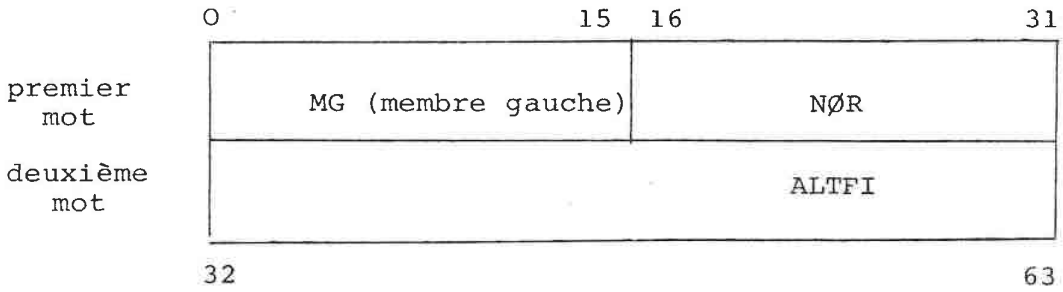
Les éléments du TGA qui ne sont pas des éléments FI auront la structure suivante :



ALT = 1er demi-mot ; contient l'adresse relative d'une alternative s'il y en a une ; zéro sinon (adresse relative à TGA - 1).

NDI = 2ème demi-mot ; contient le NDI du symbole (terminal ou non terminal).

Les éléments FI ont la structure suivante :



MG contient le NDI du membre de gauche sur un demi-mot.

NØR = 2ème demi-mot ; contient le numéro de règle ou - 1 en l'absence d'un numéro de règle.

ALTFI contient l'adresse de l'alternative éventuelle ou 0 s'il n'y en a pas.

On distingue facilement un élément FI d'un autre élément par son premier demi-mot :

- le premier demi-mot d'un élément normal est toujours positif ou nul
- le premier demi-mot d'un élément FI est toujours négatif.

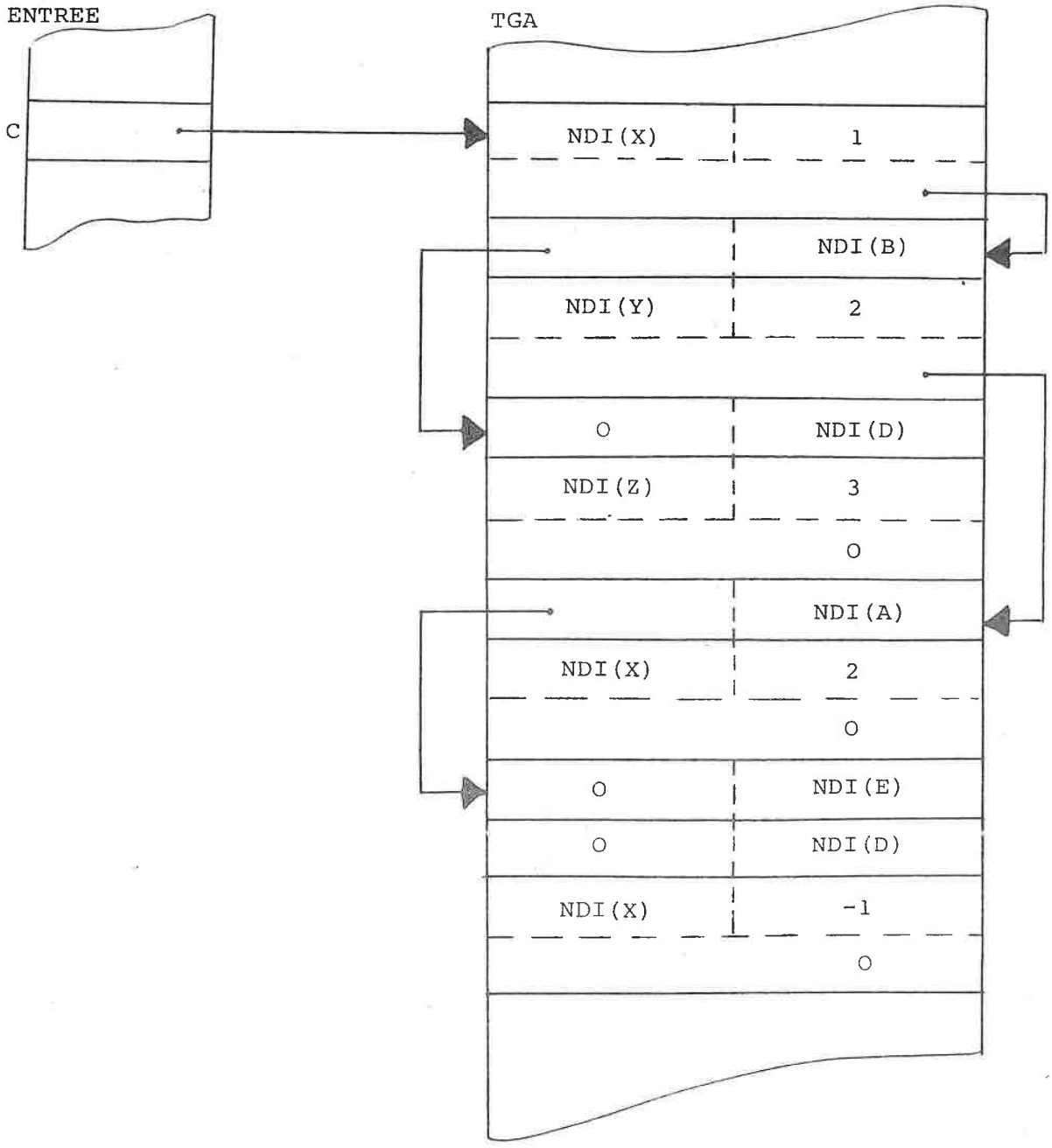
Dans une suite d'alternatives, celles qui sont réduites à un élément FI précédent toujours les autres.

Voyons sur un exemple particulier l'allure de TGA :

Supposons que dans une grammaire nous ayons les règles suivantes :

X ⟶ ABC/2/DEBC//C/1/
Y ⟶ BC/2/
Z ⟶ DC/3/E//

Pour l'entrée correspondant au symbole C le TGA aura la structure suivante :



- Les matrices de précédence

Les relations de précédence sont données par deux matrices de base et des tables complémentaires.

Les deux matrices de base donnent les relations que l'on peut avoir entre n'importe quel symbole (pouvant se trouver au sommet de la pile d'analyse) et un terminal (pouvant être lu).

La première matrice donne les relations de précédence entre les non terminaux et les terminaux. Sa dimension est :

PRECED1[-LTMS : -1 , 0 : LTDT]

La dernière colonne est celle du butoir de fin ⊢— générée par la rencontre de la fin de fichier au cours de l'analyse. Ce symbole ne doit jamais être empilé mais provoquer des réductions puis l'arrêt de l'analyse lorsque l'on a sur la pile le butoir de début puis l'axiome. Les éléments de cette colonne seront initialisés par la relation $\gtrdot$; ce symbole provoque des réductions jusqu'à avoir sur la pile —⊣ et l'axiome.

La deuxième matrice donne les relations de précédence entre les non terminaux et les terminaux. Sa dimension est :

PRECED2[0 : LTDT, 0 : LTDT]

La dernière ligne est celle du butoir de début —⊣ qui est tel que —⊣ $\lessdot$ c $\Leftrightarrow$ X $\exists^{+}$c (X étant l'axiome). Ce symbole est empilé au début de l'analyse. Cette ligne puisqu'elle dépend de l'axiome ne peut être calculée qu'au moment de l'analyse. Elle est mise à zéro par FADA

Chaque élément de la matrice PRECED1 est constitué de 3 bits.

Chaque élément de la matrice PRECED2 est constitué de 2 bits. Ils ont même signification que les 2 derniers de PRECED1.

Etant donné $A \in N \cup T$ et $c \in T$ appelons :

rel(A,c) la valeur de

si $A \in N$ alors les deux derniers bits de PRECED1[A,c]

sinon PRECED2[A,c]

rel(A,c) peut avoir les valeurs 0, 1, 2 ou 3 .

$rel(A,c) = 0 \Longleftrightarrow$ pas de relation de précédence entre A et c

$rel(A,c) = 1 \Longleftrightarrow A \lessdot c \Longleftrightarrow A \mathcal{F} \mathcal{G}^* c$

$rel(A,c) = 2 \Longrightarrow A \gtrdot c \Longrightarrow A \mathcal{F}^{-1+} \mathcal{F} \mathcal{G}^* c$

$rel(A,c) = 3 \Longrightarrow A \gtrdot c \Longrightarrow \exists D \in N$ tel que $A \lessdot D$, D vs c et $D ::= \Lambda$

La distinction entre les deux cas où la relation $\gtrdot$ existe (valeurs 3 et 4) permet au moment de l'analyse de choisir entre la réduction d'une règle dont le membre de droite est au sommet de la pile et la réduction d'une règle vide.

si $A \in N$ appelons r1(A,c) la valeur du 1er bit de PRECED1[A,c]

r1(A,c) = 0 ou 1

$r1(A,c) = 1 \Longrightarrow$ $A1 \in N$, $\exists \lambda 1$ et $\lambda \in (N \cup T)^*$ tels que $((A ::= \lambda$ et $A1 ::= \lambda 1 \lambda)$ ou $(A1 ::= \lambda$ et $A ::= \lambda 1 \lambda))$ et $c \in vs(A) \cap vs(A1)$

Dans ce cas, la connaissance au moment de l'analyse du symbole B au sommet de la pile et du terminal c qui se présente dans le texte source ne permet pas le choix de la règle. Ce choix peut être rendu déterministe par l'examen de la relation de précédence existant entre le membre de gauche de la règle et le symbole précédent sur la pile d'analyse.

Pour cela, il faut connaître la relation $\lessdot$ non seulement sur les terminaux mais aussi sur les non terminaux.

r1(A,c) = 1 signifie aussi que l'on peut trouver grâce à la table complémentaire ENTSEC l'ensemble des symboles B tels que $B \in N \cup T$ et $B \lessdot A$. ($B \mathcal{F} \mathcal{G}^* A$).

- Compléments à la matrice de précédence

Les compléments portent uniquement sur les non terminaux A tels qu'il existe c $\in$ T tel que rl(A,c) = 1.

Ils donnent alors l'ensemble des B $\in$ NuT tels que B $\prec$ A. Il s'agit donc théoriquement d'une matrice de dimension :

[- LTMS : 0 , - LTMS : LTDT]

On ne représente dans cette matrice que les lignes utiles. Pratiquement ces compléments sont donnés par une table :

ENTSEC -LTMS : 0 comportant un mot par élément.
ENTSEC[A] = 0 $\Longleftrightarrow \forall c \in T$ rl(A,c) = 0.
si ENTSEC[A] $\neq$ 0 sa valeur est l'adresse relative (par rapport à ENTSEC) de la ligne correspondante de la matrice.

Chaque ligne contient un certain nombre de mots , chacun contenant le NDI d'un symbole B tel que B $\prec$ A. Ces NDI sont triés dans chaque ligne par ordre croissant.

Le premier mot de chaque ligne indique le nombre d'éléments de la ligne.

On notera que le butoir de début de NDI : LTDT peut faire partie d'une ligne. Mais cela dépend de l'axiome choisi pour l'analyse et on ne peut donc le décider qu'au moment de l'analyse. Pour cette raison on prévoit dans chaque ligne un mot supplémentaire, non inclus dans le compte de début qui pourra recevoir au moment de l'analyse la valeur LTDT.

Au moment de l'analyse il faut compléter les matrices de précédence en fonction de l'axiome. Cela se fera dans une première partie d'initialisations.

Il faut en réalité déterminer les relations du butoir de début avec les autres symboles.

Si X est l'axiome, on pose $\longrightarrow \mathcal{S}\ X$
Il en résulte que :

$$\longrightarrow \prec B \iff \longrightarrow \mathcal{S}\ \mathcal{J}^* B \iff X\ \mathcal{J}^* B$$

$\nexists$ B tel que $\longrightarrow \succ B$.

Ainsi il faut remplir la dernière ligne de PRECED 2 avec la ligne de INT correspondant à l'axiome.

De plus un mot contenant LTDT est ajouté aux lignes de ENTSEC telles que ENTSEC[i]$\neq$ 0 et IN[X,i] = 1.

Pour calculer ces relations l'analyseur devra donc disposer des matrices IN , INT. Ces matrices seront donc incluses dans la représentation de la grammaire sous forme ascendante. L'analyseur les exploitera dans une phase d'initialisation et peut ensuite réutiliser la mémoire qu'elles occupent.

6.2. FORMAT GENERAL D'UNE GRAMMAIRE EN FORMAT ASCENDANT.

Le format de la grammaire tel qu'il doit être placé sur disque par FADA (en un ou plusieurs enregistrements) comporte les diverses tables du paragraphe précédent ainsi que la table des caractères à sauter. Ces tables y figurent dans l'ordre suivant :

- vecteur de renseignements de taille identique à celui du format de la grammaire descendante.
- la liste des caractères à sauter (précédée d'un mot en indiquant le nombre ; cette table est identique dans les deux formats de grammaire).
- table des terminaux.
- liste des vecteurs de renseignements des matrices IN, FIN, PRECED1, PRECED2.
- l'adresse relative de ENTSEC et la taille totale du complément aux matrices de précédence (éventuellement zéro).
- le TGA.

- les matrices IN, INT, PRECED1, PRECED2.
- la table ENTREE.
- la table ENTSEC et les compléments à la matrice de précédence (éventuellement).

Le vecteur de renseignements du début a la structure suivante :

mot 0 : réservé pour indiquer le type de la grammaire
mot 1 : adresse relative de la table ENTREE
mot 2 : nombre de symboles non terminaux (LTMS)
mot 3 : adresse relative du TGA
mot 4 : longueur du TGA (LTGA)
mot 5 : adresse relative de la table des terminaux
mot 6 : longueur de la table des terminaux (LTDT)
mots 7, 8, 9 et 10 : vecteur de renseignements du tableau INT.

Le vecteur de renseignements de chacune des 5 matrices de bits occupe 4 mots :

1er mot : adresse relative du tableau dans la grammaire
2ème mot: nombre de mots occupé par le tableau
3ème mot: K1
4ème mot: K2

6.3 ORGANISATION GENERALE DE L'ETUDE EN VUE DE L'ANALYSE ASCENDANTE.

FADA commence par faire l'étude de la grammaire en vue d'une analyse descendante (chap. 4).

Lorsque cette étude est terminée, on doit choisir entre 3 solutions :

- l'étude est satisfaisante et il n'y a plus qu'à créer le fichier disque contenant la grammaire.
- l'étude n'est pas satisfaisante et on entreprend une étude en vue de l'analyse ascendante.
- l'étude n'est pas satisfaisante et on abandonne le travail ou FADA choisit une troisième méthode d'analyse.

La décision sera prise dans le module de commande FADA après l'appel du sous-programme TESTSAUT suivant la valeur du niveau d'erreur NIVERR assigné à la grammaire.

- si NIVERR $\prec$ 3 alors l'étude est satisfaisante (il n'y a pas d'infraction aux règles (D'1) et (D'2)) et il suffit d'effectuer la création du fichier disque (appel des sous-programmes FIFI2, FINAL et ECRIRE (cf. chap. 4)).

- si NIVERR = 5 on sait qu'une analyse correcte est impossible puisqu'il y a une erreur sur les caractères à sauter (cf. 4.7). Cependant s'il y a eu aussi des erreurs du niveau 3 ou 4 on peut entreprendre une étude de la grammaire à titre indicatif pour savoir si après correction des erreurs sur les caractères à sauter une analyse ascendante serait possible.

- si NIVERR = 3 ou 4 l'étude de la grammaire en vue d'une analyse ascendante doit être entreprise car des infractions aux règles de déterminisme (D'1) et (D'2) ont été détectées. Cette étude se fait selon l'ossature suivante, après remise à zéro du niveau d'erreur NIVERR :

1) Etude des conflits de précédence afin de savoir si la grammaire satisfait à la première règle de déterminisme.

On dispose des matrices de précédence PRECED1 et PRECED2 construites par le sous-programme VOISEVAL.

A ce niveau les matrices PRECED1 et PRECED2 ne contiennent que les relations $\prec$ et $\mathcal{F}^{1+}$ $\mathcal{J}$ $\mathcal{J}^*$. Ce sont les conflits entre ces deux relations qui sont étudiés. La relation $\succ$ sera complétée ultérieurement et les conflits que cela peut entraîner seront étudiés en même temps.

Lorsqu'un conflit est rencontré, NIVERR prend la valeur 3. Si NIVERR = 3 après cette étude, il est inutile de poursuivre plus loin. On peut tenter une autre méthode d'analyse.

Si aucun conflit n'a été détecté l'étude se poursuit par la construction de la nouvelle représentation de la grammaire.

2) Construction de la nouvelle représentation : sous-programme RETOURN.

Ce sous-programme doit restituer à la mémoire libre la place occupée par le tableau VOISIN devenu inutile.

Puis il effectue la construction du TGA et le remplissage du tableau ENTREE. La construction du TGA se fait dans une zone de taille fixe réservée dynamiquement. Si cette zone déborde le TGA sera placé en partie sur un fichier disque de travail.

La construction du TGA s'accompagne de la complémentation de la relation $>$ en fonction des règles vides qui est rangée dans les matrices PRECED1 et PRECED2 avec test de conflits (un conflit entraîne le positionnement de NIVERR à 3).

Après cette construction le travail est abandonné si des conflits de précédence sont apparus. Sinon il faut effectuer une réorganisation de la mémoire.

3) Réorganisation de la mémoire (REORG).

Il faut réorganiser la mémoire de manière à obtenir une organisation proche de celle que l'on veut obtenir sur le fichier disque (décrite précédemment).

- Le vecteur de renseignement doit être modifié.

- Il faut placer les vecteurs de renseignement de IN, FIN, PRECED1 et PRECED2 après la table des terminaux.

- le TGA va remplacer le GRG. Il faut déplacer les matrices IN, FIN, PRECED1 et PRECED2 à la suite du TGA (celui-ci est en général plus court que le GRG).

4) Contrôle de la grammaire (VERIFA).

Contrôle de la grammaire obtenue selon les règles de déterminisme (D3), (D4) et (D5) (en cas de conflit NIVERR = 4). Eventuellement les compléments aux matrices de précédence sont calculés.

De même qu'au cours de l'étude de la grammaire en vue de l'analyse descendante, les infractions aux règles de déterminisme (D3), (D4) et (D5) provoquent l'appel de sous-programmes spécifiques. Ces sous-programmes peuvent, soit émettre des messages d'erreur précisant les numéros des règles (les numéros des lignes du programme FACE ont disparus) en infraction, soit comptabiliser ces erreurs, si une autre méthode d'analyse peut être envisagée.

Après ce contrôle le traitement peut s'arrêter si NIVERR = 3. Sinon on va pouvoir placer la grammaire sur disque.

5) Rangement de la grammaire sur disque.

o
o o

Chacune de ces 5 parties constituera un sous-programme appelé par le module de commande FADA.

Chacun de ces cinq sous-programmes va maintenant être étudié.

6.4. ETUDE DES CONFLITS DE PRECEDENCE : LE SOUS-PROGRAMME PRECEDEN.

Ce sous-programme a pour rôle d'examiner chaque élément des matrices PRECED1 et PRECED2, et de détecter les conflits, c'est-à-dire les éléments égaux à 3.

En cas de conflit le sous-programme PRECEDER est appelé. PRECEDER positionne le niveau d'erreur de la grammaire NIVERR à la valeur 3 . Dans sa première version PRECEDER émet un message d'erreur indiquant entre quels symboles un conflit a été détecté. Dans ce message les symboles non terminaux apparaissent sous forme de l'opposé du NDI. Ce numéro correspond à l'ordre d'apparition du non terminal en tant que membre gauche dans le

programme FACE. Les terminaux du type chaîne de caractères apparaissent en clair. Les terminaux du type langage régulier apparaissent sous la forme de 3 caractères débutant un mot du langage.

Une deuxième version du sous-programme PRECEDER est nécessaire si l'on dispose d'une méthode d'analyse capable d'accepter des conflits de précédence. Le sous-programme PRECEDER ne doit plus émettre de message d'erreur, mais éventuellement conserver des informations sur les conflits.

6.5. CONSTRUCTION DE LA NOUVELLE REPRESENTATION DE LA GRAMMAIRE : LE SOUS-PROGRAMME RETØURN

RETØURN génère dans le TGA toutes les règles se terminant par un symbole donné et ceci pour chaque symbole (terminal ou non terminal).

Pour chaque symbole RETØURN détermine les règles qui l'ont comme finale. Cette détermination se fait à l'aide de la relation $\mathcal{F}$. Cette relation se trouve dans le tableau FINALE, tableau construit par le sous-programme FINSAV appelé immédiatement après 2PASSAGE. Le rôle de FINSAV est simplement de recopier le tableau FIN dans FINALE avant que la relation $\mathcal{F}$ contenue dans FIN ne soit transformée en $\mathcal{F}^+$ par WARSHALL.
Grâce à la relation $\mathcal{F}$, RETØURN peut faire correspondre à tout symbole s, la liste des non terminaux X, membres de gauche d'une règle se terminant par s.

A l'aide du sous-programme PARCØURS, RETØURN peut localiser parmi les règles de réécriture d'un symbole X, celles qui se terminent par le symbole s.

Lorsque PARCØURS donne accès à un élément final référençant s, on peut construire la représentation inverse de la règle à laquelle s appartient en suivant les chaînages du GRG puisque PARCØURS a inversé les liens de chaînage.

L'architecture du sous-programme RETØURN se déduit de son principe. Pour chaque symbole s, RETØURN demande à PARCØURS

de parcourir les règles dont s est susceptible d'être finale. PARCØURS appelle le sous-programme CØNSTAT à la rencontre de chaque symbole.

Lorsque CØNSTAT constate l'apparition du symbole s suivi d'un élément FI, FI1 ou FI2 il active le module CØNSTRUIRE chargé de construire la nouvelle représentation de la grammaire.

La construction du TGA se fait dans un buffer ZBUF réservé dynamiquement de LENR mots. Lorsque ce buffer est plein, le TGA est placé sur disque, dans un fichier de travail. Au cours de la construction du TGA deux indices sont entretenus :

IB indice relatif à ZBUF variant de 0 à LENR ;
IR indice réel dans le TGA (nombre de mots du TGA).

Dans le TGA, les éléments FI doivent figurer en tête d'une liste d'alternatives. Cela peut entraîner des réorganisations partielles du TGA au cours de sa construction. Pour que ces réorganisations puissent se faire, il est nécessaire que toutes les règles dépendant de l'élément de la table ENTREE en cours de construction se trouvent en mémoire et non dans le fichier de travail. Aussi un autre indice relatif à ZBUF est entretenu : IK.

IB indique toujours l'adresse du premier mot libre dans ZBUF ;
IK indique l'adresse du premier mot devant être conservé en mémoire, même si une écriture disque est nécessaire.

Le sous-programme RETØURN s'écrit :

```
début libération de l'espace mémoire occupé par le tableau VOISIN ;
      réservation du tableau ENTREE et du buffer ZBUF ;
      IB:=IK:=0 ; IR:=1 co les indices réels commencent à 1 co ;
      ETAP  CØNSTAT  co préparation de PARCØURS co ;
      pour s:=-LTMS pas 1 jusqua LTDT faire
      début pour j:=-LTMS pas 1 jusqua -1 faire
            si FINALE[j,s] alors PARCØURS(TMS[j]) ;
```

```
            IK:=IB co mise à jour de IK avant de passer à l'élément
                    suivant de ENTREE co
        fin ;
        co la construction du TGA est presque terminée ; il ne reste
           plus qu'à y adjoindre les règles vides co
        pour j:=-LTMS pas 1 jusqua -1 faire
        début examin:=TMS[j] ;
            voir:si TFRE[examin]=FI alors
                        début si ENTREE[LTDT]=0 alors ENTREE[LTDT]:=IR
                                    co ENTREE[LTDT]= entrée des
                                            règles vides co
                                sinon ZBUF[IB-1]:=IR co
                                        chaînage d'alternative
                                        de l'élément FI précé-
                                        dent co ;
                                si IB ⩾ LENR -2 alors TASSER ;
                                1er demi-mot de ZBUF[IB]:=j ;
                                2ème demi-mot de ZBUF[IB]:=IDENT[examin];
                                ZBUF[IB+1]:=0 ;
                                IB:=IB+2 ; IR:=IR+2 ;
                                CØMPVID co sous-programme complétant la
                                           relation > co
                        fin
                                sinon
                        début examin:=ALT[examin] ;
                                si examin ≠ 0 alors allera voir
                        fin
        fin
fin
```

Lorsque CØNSTAT est appelé par PARCØURS, il reçoit :
dans RPX2 l'adresse d'un élément ST ou SNT ou FI1 ;
dans RPX1 en général l'adresse de l'élément précédent ;
dans AD1ELMT le paramètre de PARCØURS.

La situation dans le GRG de l'élément dont l'adresse se trouve dans RPX2 dépend du point d'appel de CØNSTAT à l'intérieur de PARCØURS.

UNIVERSITE NANCY

Le point d'appel SPN1 correspond à la rencontre du premier élément (RPX2 = AD1ELMT).

Le point d'appel SPN2 correspond à un élément rencontré en suivant le lien "suc". RPX1 contient alors l'adresse de l'élément précédent.

Le point d'appel SPN3 correspond à l'examen du premier élément d'une nouvelle alternative. RPX1 contient alors l'adresse de l'élément ayant donné accès à cette alternative.

Le sous-programme CØNSTAT (ou CØNSTRUIRE appelé par CØNSTAT) a la possibilité de connaître son point d'appel en comparant la valeur du registre de liaison à SPN1, SPN2 ou SPN3.

Pratiquement le sous-programme CØNSTAT ne sert qu'à décider de l'appel du sous-programme CØNSTRUIRE :

```
début RX3:=TFRE[RPX2] ;
      R4:=NDI[RPX2] ;
      si RX3 ≠ FI1 et R4 = s alors
                  début R4:=SUC[RPX2] ;
                  autre: RX3:=TFRE[RPX2] ;
                      si RX3=FI ou RX3=FI1 ou RX3=FI2 alors CØNSTRUIRE
                          sinon début R4:=ALT[R4] ;
                                      si R4≠0 alors allera autre
                                fin
                  fin
fin
```

Le sous-programme CØNSTRUIRE suit les prédécesseurs de l'élément d'adresse RPX2 et les intègre dans le TGA. Cette intégration se fait en consultant les règles déjà construites pour le même symbole afin de mettre en commun les fins de règle identiques.

```
début norègle:=IDENT[R4] co mise en réserve du n° de règle pour la
                            construction de l'élément FI co ;
      examin:=IK ;
```

```
si ENTREE[s]=0 alors ENTREE[s]:=IR co cas où il s'agit de
        la première règle se terminant par le symbole s co ;
si le point d'appel de CØNSTAT est SPN3 alors co RPX2 pointe
        vers le début d'une nouvelle alternative du GRG ;
        pour retrouver le prédécesseur il faut remonter la
        suite des alternatives co
        début remont:si l'élément repéré par RPX1 a été marqué
                        par PARCØURS alors début RPX1:=SUC[RPX1]
                                                co RPX1:=alterna-
                                                  tive précédente
                                                  co ;
                                                  allera remont
                                              fin ;
                     co on est en présence du premier élément
                        de la liste d'alternatives. On peut
                        donc accéder au prédécesseur à moins
                        que RPX1 ne pointe vers le premier
                        élément de la première règle de réécri-
                        ture du non terminal j co
                     si RPX1=AD1ELMT alors RPX1:=0
                          sinon RPX1:=SUC[RPX1] ;
                     allera rangement ;
        fin ;
si le point d'appel de CØNSTAT est SPN1 alors RPX1:=0 ;
rangement:co dans tous les cas RPX1 contient maintenant l'adresse
        du prédécesseur de RPX2 ou 0 si l'élément examiné est
        un début d'une règle co
        si RPX1≠0 alors début RX3:=TFRE[RPX1] ;
                              1er demi-mot de ITEM:=0 ;
                              2ème demi-mot de ITEM:=si RX3=FI1
                                                       alors j
                                                       sinon NDI[RPX1];
                           co ITEM contient un élément qui doit
                              être intégré au TGA ; dans le cas où
                              le prédécesseur rencontré est FI1, la
                              récursivité à gauche est rétablie co
```

```
                    MASK=X'FFFF' co ce masque permet de
                         comparer ITEM à des éléments du
                         TGA en ne tenant compte que du
                         deuxième demi-mot contenant le
                         NDI et non du premier (le lien
                         "alt") co
                fin
            sinon co création d'un élément FI co
                début 1er demi-mot de ITEM:=j co NDI du
                         membre de gauche co ;
                      2ème demi-mot de ITEM:= norègle ;
                      MASK:=X'FFFF0000' co ce masque
                         permet de comparer ITEM à des
                         éléments du TGA en ne tenant
                         compte que du premier demi-mot co
                fin ;
        co un élément du TGA est maintenant construit dans ITEM.
           Il faut placer cet élément dans ZBUF. Pour mettre en
           commun les fins de règle identiques on compare cet
           élément avec l'élément de ZBUF repéré par examin.
           A la création d'une nouvelle règle (début de CØNSTRUIRE)
           examin désigne le début des règles dépendant de l'entrée
           s  co
        si examin=IB alors co le contenu de ITEM doit effectivement
           être ajouté au TGA co
                ajout:début si IB ⩾ LENR-2 alors TASSER ;
                         ZBUF[IB]:=ITEM ;
                         examin:=IB:=IB+1 ; IR:=IR+1 ;
                         si RPX1=0 alors co il s'agit d'un
                                           élément FI de 2
                                           mots co
                                      début ZBUF[IB]:=0 ;
                                           examin:=IB:=IB+1 ;
                                           IR:=IR+1
                                      fin ;
                         allera suite
                fin
voyons:si(ZBUF[examin] et MASK) = ITEM alors
        co on est en présence d'un élément identique à celui
```

```
        que l'on veut placer dans le TGA co
        début examin:=examin+1;
              allera suite
        fin ;
     co l'élément désigné par examin est différent de ITEM ;
        il faut regarder les autres alternatives co
        dm1:=typ:=premier demi-mot de ZBUF[examin] ;
        si dm1 < 0 alors dm1:=ZBUF[examin+1] ;
        co dm1=lien"alt"de l'élément ZBUF[examin] co
        si dm1=0 alors début co il n'y a pas d'autre alter-
                                native à examiner ; il faut
                                en créer une co
                             si typ < 0 alors ZBUF[examin+1]:=IR
                                   sinon si RPX1 < 0
                                     alors allera addfi
                                     sinon ler demi-mot de
                                       ZBUF[examin]:=IR ;
                             allera ajout
                          fin ;
        si RPX1≠0 ou typ < 0 alors co la nouvelle alternative
                                     doit être examinée co
                                début examin:=dm1 ;
                                      allera voyons
                                fin ;
addfi: co un élément FI doit être ajouté ; il doit précéder les
        autres alternatives déjà présentes  ; il est donc né-
        cessaire de les déplacer de deux positions vers l'avant
        co
        si IB ≥ LENR-2 alors TASSER
        pour ℓ:=IB+1 pas -1 jusqua examin+1 faire
        ZBUF[ℓ]:=ZBUF[ℓ-2] ;
        IB:=IB+2 ; IR:=IR+2 ;
        ZBUF[examin]:=ITEM ;
        ZBUF[examin+1]:=examin+2+IR-IB co lien"alt"co ;
        pour ℓ:=IK pas 1 jusqua IB-1 faire co mise à jour des
                                             pointeurs co
        début dm1:=ler demi-mot de ZBUF[ℓ] ;
              si dm1 < 0 alors début ℓ:=ℓ+1 ;
                                   si ZBUF[ℓ] > examin+IR-IB
                                     alors ZBUF[ℓ]:=ZBUF[ℓ]+2
                              fin
```

```
                        sinon si dml > examin+IR-IB
                                alors 1er demi-mot de
                                      ZBUF[ℓ]:=dml+2
            fin ;
            allera retour ;
suite : co un élément d'une règle vient d'être intégré au TGA.
        Il faut passer à l'élément précédent dans le GRG co
        si RPX1≠0 alors
           début si RX3 = FI1 alors RPX1:=0
                              sinon autre :
                                début si l'élément du GRG
                                       repéré par RPX1 est
                                       marqué par PARCØURS
                                       alors
                                         début RPX1:=SUC[RPX1]
                                                allera autre
                                                co remontée
                                                d'une suite
                                                d'alternatives
                                                co
                                          fin ;
                                     si RPX1=AD1ELMT
                                        alors RPX1:=0
                                        sinon RPX1:=SUC[RPX1]
                                fin ;
                    allera rangement
                fin ;
retour: fin
```

Le sous-programme TASSER est appelé par CØNSTRUIRE pour libérer de la place dans la zone tampon ZBUF. Pour cela TASSER place dans le fichier de travail le début de la zone ZBUF (jusqu'à l'indice IK exclu) ; puis le reste de ZBUF est déplacé au début.

```
début si IK ≤ 1 alors début émettre un message de débordement
                            mémoire ; abandon du travail
                     fin ;
    si le fichier de travail est fermé alors l'ouvrir ;
```

```
      écrire l'enregistrement correspondant à ZBUF[O:IK-1].
      co le fichier doit être à enregistrements de longueur variable
         co ;
      t:=IB-IK-1 ;
      s:=IB-examin ;
      mouvementer ZBUF[IK:IB-1] en ZBUF[O:t] ;
      IK:=O ;
      IB:=t+1 ;
      examin:=IB-s
fin
```

Pour terminer l'étude de RETØURN et des sous-programmes qu'il utilise, il reste à examiner le sous-programme CØMPVID.

Ce sous-programme ne participe pas à la construction de la représentation inverse de la grammaire, mais à la complémentation de la relation de précédence $\gtrdot$ en fonction des règles vides. Il ne fait donc pas logiquement partie de RETØURN. Il en fait partie pour une simple raison d'efficacité : RETØURN détecte les règles vides et CØMPVID profite de cette détection.

CØMPVID est appelé par RETØURN à la rencontre de chaque règle vide et a alors accès à la variable j de RETØURN qui contient le NDI d'un métasymbole membre gauche d'une règle vide. CØMPVID doit compléter la relation $\gtrdot$ pour tous les couples (A,c) $A \in N \cup T$ et $c \in T$ tels que $A \lessdot D$ et D vs c. CØMPVID ne dispose plus du tableau VØISIN contenant la relation vs, mais il retrouve cette relation dans les matrices de précédence puisque vs = $\lessdot \cup (\gtrdot$ non complétée). La relation $\lessdot$ avec le non terminal j doit être recherchée à l'aide de VDR et IN.

```
début pour il:=-LTMS pas 1 jusqua -1 faire
      début pour k:=-LTMS pas 1 jusqua -1 faire
            si VDR[il,k] et IN[k,j] alors
              début co on a la relation il ≺ j co
                   pour jl=O pas 1 jusqua LTDT-1 faire
                    si PRECED1[j,jl]=1 ou PRECED1[j,jl]= 2
                      alors co il ≺ j et j vs jl co
```

```
                        début si PRECED1[i1,j1]= 1 ou 2
                                alors co conflit de précédence co
                                      PRECEDER(i1,j1)
                                sinon PRECED1[i1,j1]:=3 ;
                        fin ;
                    allera kfin1
              fin ;
      kfin1:fin ;
            pour i1:=0 pas 1 jusqua LTDT-1 faire
            début pour k:=-LTMS pas 1 jusqua -1 faire
                  si VDR[i1,k] et IN[k,j] alors
                     début co on a la relation i1 ≺ j co
                           pour j1:=0 pas 1 jusqua LTDT-1 faire
                           si PRECED1[j,j1]=1 ou PRECED1[j,j1]=2
                              alors co i1 ≺ j et j vs j1 co
                                   début si PRECED2[i1,j1]=1 ou 2
                                           alors co conflit de
                                                    précédence co
                                                 PRECEDER(i1,j1)
                                           sinon PRECED2[i1,j1]:=3
                                   fin ;
                              allera kfin2.
                     fin;
      kfin2:fin
fin
```

6.6. REORGANISATION DE LA MEMOIRE : LE SOUS-PROGRAMME REØRG.

REØRG doit placer le TGA en mémoire à la place du GRG et placer à sa suite les tableaux :
INT IN FIN PRECED1 PRECED2 ENTREE et VDR.
Ces sept tables devront être déplacées en mémoire en général vers la basse mémoire (car le plus souvent le TGA est sensiblement plus court que le GRG), éventuellement vers la haute mémoire.

Afin de manipuler plus commodément ces 7 tables on crée un descriptif de 14 mots DESTAB où l'on trouve 2 mots pour chacune de ces tables dans l'ordre d'occupation de la mémoire. Le premier mot donne l'adresse de début de la table. Le second donne la taille en mots de la table.

Après une séquence d'initialisations, on teste la taille de l'espace libre avant le tableau INT. Si cet espace est suffisant pour recevoir le TGA on réorganise la mémoire au moyen du module KNORMAL. Sinon on réorganise la mémoire au moyen du module KSPEC.

En principe le module KSPEC doit être rarement utilisé : on peut envisager de le remplacer dans un premier temps par un abandon du travail et de ne le programmer que lorsque le besoin s'en fera réellement sentir.

On trouvera une description complète de REØRG en annexe 3.

6.7. CONTROLE DE LA GRAMMAIRE : LE SOUS-PROGRAMME VERIFA.

Le sous-programme VERIFA doit contrôler la grammaire en fonction des règles (D3), (D4) et (D5).

Ce contrôle se fait en deux temps :

Dans un premier temps on examine la grammaire en fonction des règles (D3) et (D4).

Dans un deuxième temps on contrôle la grammaire pour la règle (D5), ce qui se fait en utilisant uniquement les tableaux de précédence et la table des terminaux.

1. Première partie de VERIFA.

La première partie du sous-programme VERIFA contrôle la grammaire en fonction des règles de déterminisme (D3) et (D4). En même temps les compléments aux matrices de précédence sont construits.

Le principe de l'algorithme consiste à construire un squelette des groupes de règles se terminant par la même partie droite. Ce squelette est ensuite étudié par le sous-programme ETUDE.

Le squelette est construit dans le tableau SQLT qui fonctionne comme une pile. Ce squelette comporte les éléments FI des règles ayant une partie droite facteur droit d'une autre. Ainsi pour les règles A1::=λ/1/ et A2::=ψλ/2/ on trouve empilés (A2,2) et (A1,1). Si ψ≠Λ, ces deux éléments sont séparés dans la pile par l'élément du TGA correspondant au dernier symbole de ψ. Le tableau SQLT peut utiliser l'espace qui avait été attribué aux tableaux DESTAB et NDESTAB de REØRG soit 28 mots . Le premier mot de SQLT sera un butoir contenant dans son premier demi-mot un nombre strictement positif et dans son deuxième demi-mot un nombre positif très grand.

```
début réservation du tableau ENTSEC qui est initialisé à zéro ;
      AR:=LTMS co adresse relative à ENTSEC du 1er mot libre dans
                  la représentation du complément aux matrices de
                  précédence co ;
      initialisation SQLT(0) ; hℓ:=27 co valeur limite de la hauteur
                                      de SQLT co ;
     pour i:=-LTMS pas 1 jusqua LTDT faire
     début RX1:=ENTREE[i] ;
           si RX1≠0 alors
             début co RX1 est l'adresse relative de règles se
                      terminant par le symbole i. Ces règles vont
                      être parcourues afin de construire leur
                      squelette dans SQLT co
                      h:=0 co hauteur de la pile SQLT co ;
                      deb:si 1er demi-mot de TGA[RX1] ≥ 0
                           alors co l'élément repéré par RX1 est
                                   un symbole ; on passe à l'élément
                                   suivant en séquence co
                                   début RX1:=RX1+1 ;
                                        allera deb ;
                                   fin ;
                   co on est en présence d'un élément FI à
                      empiler co
```

```
demfi :  h:=h+1 ;
      si h > hℓ alors début émettre un message de
                                débordement ;
                                abandon
                           fin ;
      SQLT[h]:=TGA[RX1] ;
      RX2:=TGA[RX1+1] co RX2=champ ALT de l'élément
                         FI co
      si RX2 ≠ 0 alors co l'élément FI conduit à une
                         liste d'alternatives qu'il
                         faut examiner co
        alt:début ler demi-mot de TGA[RX2-1]:=-RX1
                co l'élément précédent l'élément
                   RX2 est forcément un élément FI.
                   Son 3ème demi-mot qui est libre
                   est utilisé pour construire un
                   chaînage arrière vers l'alterna-
                   tive que l'on quitte co ;
                   RX1:=RX2 ;
                si ler demi-mot de TGA[RX1]<0 alors
                   co la nouvelle alternative est
                      un élément FI co
                      allera elemfi ;
                co la nouvelle alternative est un
                   symbole qu'il faut empiler co
                h:=h+1 ;
                si h > hℓ alors début émettre un
                                      message de
                                      débordement ;
                                      abandon
                                   fin ;
                SQLT[h]:=TGA[RX1] ;
                RX1:=RX1+1 co passage au successeur co
                allera deb
             fin ;
      co RX2=0 il n'y a pas d'autre alternative ; il faut
         étudier le squelette construit puis passer à un
         autre co
         ETUDE ;
```

```
        depil: h:=h-1 ;
               si 1er demi-mot de SQLT[h]≤ 0
                 alors allera depil ;
               co la pile SQLT est dépilée jusqu'à un
                  éventuel symbole qui peut avoir d'autres
                  alternatives. On remonte ensuite dans le
                  TGA en examinant les nouvelles alterna-
                  tives possibles co
     razremont : indic:=-1 ;
        remont: si RX1=ENTREE[i] alors allera finpour ;
                RX1:=RX1-1 ;
                RX2:=1er demi-mot de TGA[RX1];
                si RX2 < 0 alors co rencontre d'un chaînage
                                    arrière co
                           début 1er demi-mot de TGA[RX1]:=0
                                 co RAZ lien de chaînage co ;
                                 si indic=-1
                                    alors début RX1:= -RX2 ;
                                                allera remont ;
                                          fin ;
                                 redepil: h:=h-1 ;
                                 si 1er demi-mot de SQLT[h]≤0
                                      alors allera redepil ;
                                 RX1:=-RX2 ;
                                 si indic≠0
                                   alors début RX2:=indic ;
                                               allera alt ;
                                         fin ;
                                 allera razremont
                           fin ;
                 co on n'est pas en présence d'un chaînage
                    arrière co
                 si indic≤0 alors début indic:=RX2 ;
                                        allera remont
                                 fin ;
                 co indic contient l'adresse d'une alternative co
                 RX1:=RX1+1 ; RX2:=indic ; allera alt
     fin
  fin
fin
```

Pour chaque squelette construit, VERIFA appelle le sous-programme ETUDE qui fait le contrôle proprement dit et construit éventuellement les compléments aux matrices de précédence. Il reçoit dans h la hauteur de la pile SQLT.

```
procédure ETUDE ;
début pour hh:=1 pas 1 jusqua h-1 faire co si h=1 il n'y a rien
                                          à faire co
    début A1:=1er demi-mot de SQLT[hh] ;
        si A1 < 0 alors co hh correspond bien à un élément FI
                           empilé co
                pour hh1:=hh pas 1 jusqua h faire
                début A2:=1er demi-mot de PILE[hh1] ;
                    si A2 < 0 alors co A2 correspond bien à un
                                       élément FI empilé et l'on
                                       a A1:=λ et A2::=ψλ  co
                        début errindic:=0 co indicateur d'infraction:
                                             0 : on ne sait pas
                                                 encore s'il y a
                                                 infraction
                                             1 : A1 et A2 ont des
                                                 voisins communs co ;
                            pour j:=0 pas 1 jusqua LTDT-1 faire
                                    co boucle d'examen de chaque
                                       terminal co
                            si PRECED1[A1,j] ≠ 0 et PRECED1[A2,j] ≠ 0
                              alors co un conflit existe entre les
                                       voisins de A1 et A2 :
                                       A1::=λ, A2::=ψλ et j ∈ vs(A1)
                                       ∩ vs(A2). Ce conflit doit être
                                       indiqué dans PRECED1[A1,j] co
                                       début 1er bit de
                                             PRECED1[A1,j]:=1 ;
                                             errindic:=1
                                       fin ;
```

```
si errindic=1
  alors co il faut essayer de lever les
          indéterminations à l'aide des com-
          pléments aux matrices de précédence co
  début si ENTSEC(A1)=0 alors CØNSEC(A1) co
          construction de l'entrée secondaire
          de A1 co ;
        pour IB:=hh+1 pas 1 jusqua hh1-1 faire
        si 1er demi-mot de SQLT[IB] ≥ 0 alors
         début B:=2ème demi-mot de SQLT[IB] ;
           co les deux règles sont de la forme
             A1::=λ  A2::=μBλ
             si B ≺ A1 il  y a infraction à la
               règle 3 ; ce test se fait à
               l'aide du sous-programme
               ACSEC co
             si ACSEC(A1,B) alors
               début no1:=2ème demi-mot de
                          SQLT[hh] ;
                     no2:=2ème demi-mot de
                          SQLT[hh1] ;
                     ERAS3(no1,no2) co sous-
                          programme de trai-
                          tement des erreurs
                          sur les numéros de
                          règles no1 et no2 co
               fin ;
             allera finexam
         fin ;
        co les deux règles ont même membre de
           droite:  A1::=λ et A2::=λ ; on ne
           doit pas avoir B tel que B ≺ A1 et
           B ≺ A2 co
        si ENTSEC(A2)=0 alors CØNSEC(A2) co
           construction de l'entrée secondaire
           de A2 co ;
```

```
                co il faut maintenant chercher s'il
                  existe B tel que B ≺ A1 et B ≺ A2 co
                  AD1:=ENTSEC[A1] ; AD2:=ENTSEC[A2] ;
                  L1:=ENTSEC[AD1] ; L2:=ENTSEC[AD2] ;
                  co adresses et longueurs des listes
                    de compléments correspondant à A1
                    et A2 co
                  jB:=1 ;
                  pour iB:=1 pas 1 jusqua L1 faire
                  début compar :
                    si jB > L2 alors allera finexam ;
                    B1:=ENTSEC[AD1,iB] ;
                    B2:=ENTSEC[AD2,jB] ;
                    si B2 < B1 alors
                              début jB:=jB+1 ;
                                    allera compar
                              fin ;
                    si B1=B2 alors co une erreur est
                                      rencontrée co
                             début no1:=2ème demi-mot
                                      de SQLT[hh] ;
                                   no2:=2ème demi-mot
                                      de SQLT[hh1] ;
                                   ERAS4(no1,no2,B1) ;
                                   iB:=iB+1
                             fin
                  fin
                fin ;
        finexam:fin
    fin
fin
```

Le sous-programme CØNSEC construit les entrées secondaires du non terminal qu'il reçoit en paramètre. Il utilise pour cela les matrices VDR et IN. La construction de ces entrées secondaires se fait en mémoire dynamique à la suite de la table d'accès ENTSEC. CØNSEC doit donc gérer la mémoire dynamique. Dans le cas où une demande de mémoire provoque l'allocation d'une nouvelle section de pile non en séquence en mémoire, il est nécessaire de translater la partie déjà construite dans la nouvelle section.

AR initialisé au début de VERIFA représente toujours l'adresse relative à ENTSEC du 1er mot libre. A la fin de VERIFA AR représente la longueur totale des entrées secondaires. Si AR=LTMS cette longueur peut être considérée comme nulle.

L'analyse détaillée de CØNSEC se trouve en annexe 4.

Le sous-programme ACSEC utilisé dans ETUDE permet un accès aux compléments des matrices de précédence. Ce sous-programme sera utilisé aussi dans l'analyseur ascendant (chap. 7).

Il reçoit deux paramètres : A et B, A est le NDI d'un symbole non terminal et B, le NDI d'un symbole quelconque.

ACSEC renvoie la valeur vrai si B $\prec$ A
faux sinon.

ACSEC ne doit bien sûr être appelé que s'il existe une entrée pour le symbole A dans les compléments aux matrices de précédence (ENTSEC[A] $\neq$ 0). S'il n'en est pas ainsi , son résultat est aléatoire.

A l'intérieur de l'entrée dépendant de A, la recherche de B peut être soit séquentielle, soit dichotomique.

Les sous-programmes ERAS3 et ERAS4 ont des rôles semblables au sous-programme PRECEDER (cf. 6.3). Ils positionnent le niveau d'erreur de la grammaire à NIV4 et dans leur première version émettent des messages d'erreur spécifiant les numéros des règles erronées. Si l'on dispose d'un analyseur acceptant ces conflits il pourront se transformer en se contentant de comptabiliser les conflits rencontrés.

2. Deuxième_partie_de_VERIFA.

La deuxième partie de VERIFA effectue une réorganisation de la mémoire, profitant de ce que le tableau VDR est maintenant inutile. Les compléments aux matrices de précédence sont translatés à la place de VDR. Un espace au moins égal à la taille du tableau

VDR peut être rendu à la mémoire libre. On trouvera cette partie du sous-programme VERIFA dans l'annexe 5.

3. Troisième_partie_de_VERIFA : Contrôle de la grammaire en fonction de la règle D5.

Le contrôle des terminaux selon la règle (D5) peut se faire avec le sous-programme EMPILER déjà employé pour le contrôle de la grammaire en vue de l'analyse descendante (cf. 4.10).

Le contrôle porte sur les familles de terminaux T(X) telles que

$$T(X) = \{c \mid c \in T \text{ et } X \mathcal{J} J^* c\}$$

pour tous les symboles X de $N \cup T$.

C H A P I T R E VII

L'analyseur ascendant ANALAS

7.1. LES DONNEES DE L'ANALYSEUR.

L'analyseur ascendant a comme données la grammaire traitée par FADA d'une part, le texte source d'autre part. Au moment où l'analyseur proprement dit est activé, la grammaire se trouve déjà en mémoire. Elle y a été placée par le module PREAMBULE commun à tous les analyseurs.

Le module PREAMBULE détermine le type d'analyseur correspondant à la grammaire et l'appelle en lui communiquant :

- l'adresse en mémoire de la grammaire (dans le mot T)
- le numéro de règle ayant l'axiome pour membre gauche, (dans le mot NRAX).

L'analyseur ascendant commence par une série d'initialisations comportant en particulier la recherche du NDI de l'axiome et la complémentation des matrices de précédence en fonction de cet axiome.

Le texte source est lu, comme pour l'analyseur descendant au moyen du module LECT décrit en annexe 2.

7.2. PRINCIPE DE L'ALGORITHME D'ANALYSE.

L'analyseur ascendant est déterministe. A chaque itération l'action entreprise dépend du symbole B se trouvant au sommet de la pile d'analyse B et du terminal c, prochain terminal à analyser lu à l'avance.

Les matrices PRECED1 et PRECED2 permettent de savoir s'il faut faire une lecture ou une réduction (ou s'il y a une erreur). Dans le cas d'une réduction le TGA et éventuellement les compléments aux matrices de précédence (ENTSEC) permettent de choisir la réduction.

La pile d'analyse contient les NDI des symboles empilés (1 mot par symbole).

Il est en général impossible de lire de façon déterministe un terminal à l'avance (un terminal pouvant être facteur gauche d'un autre). Aussi comme pour l'analyseur descendant on utilise une technique de lecture à l'avance différée. Ici on tente toujours de lire un symbole c qui soit voisin du symbole B au sommet de la pile. FADA assure que cette lecture est déterministe.

On utilise toujours le sous-programme LECT pour faire ce travail comme dans l'analyseur descendant. Ce sous-programme LECT est utilisé par l'intermédiaire du sous-programme LIRA.

LIRA possède 5 paramètres :

c = résultat = NDI du terminal lu ;

B = NDI du symbole dont on veut lire un voisin ;
B $\in$ N $\cup$ T

r1 = 0 si B $\in$ T
= 1er bit de PRECED1[B,c] si B $\in$ N

r2 = 2 derniers bits de $PRECED_2^1$[B,c]

si r2 = 0 les valeurs de c et r1 n'ont aucune signification.

Au début de l'analyse, la pile d'analyse est en principe vide ce qui provoque une difficulté pour la lecture du 1er terminal. Pour remédier à cela on place sur la pile le butoir de début ⟝ dont le NDI est LTDT. Compte tenu des relations de précédence construites on s'assure ainsi la lecture comme premier terminal d'une initiale terminale de l'axiome.

7.3. LE MODULE D'INITIALISATIONS MØDINIT2.

est chargé d'effectuer les diverses initialisations propres à l'analyseur ascendant.

ENTREE:=adresse réelle de la table ENTREE ;

LTMS = nombre de métasymboles ;

ENTSEC = adresse de la table des entrées secondaires ou zéro ;

PRECED1 et PRECED2 = groupes de 4 mots décrivant les matrices de précédence avec les adresses réelles en mémoire ;
IN et INT = groupes de 4 mots décrivant les matrices IN et INT avec les adresses réelles en mémoire ;
ATGA = adresse de début du TGA (adresse réelle) ;
LTGA = longueur du TGA (en mots).

Il faut maintenant rechercher NDIAX (NDI de l'axiome) en parcourant le TGA à la recherche d'un élément FI portant le numéro de règle NRAX.

```
pour i:=1 pas 1 jusqua LTGA faire
début a1:=1er demi-mot de TGA[i] ;
      si a1 < 0 alors début a2:=2ème demi-mot de TGA[i] ;
                            si a2=NRAX alors début NDIAX:=a1 ;
                                                   allera finch
                                              fin
                        fin
fin ;
imprimer le message: "grammaire sans axiome" ;
abandon du travail ;
finch: co on dispose du NDI de l'axiome. Il faut construire la
          dernière ligne de la matrice PRECED2 en fonction de
          l'axiome ; on a ⟶JJ* c ⟺ X J* c ⟺ INT[X,c] = 1 co
       pour i:=0 pas 1 jusqua LTDT-1 faire
            si INT[NDIAX,i]= 1 alors PRECED2[LTDT,i]:=1
       co il faut aussi compléter les entrées secondaires co
si des compléments aux matrices de précédence existent dans la
   grammaire alors
   début pour i:=0 pas 1 jusqua LTMS-1 faire
         si ENTSEC[i]≠0 alors
            début si IN[NDIAX,i]=1 alors ENTSEC[i,0]:=ENTSEC[i,0]+1
                        co il suffit d'incrémenter des compléments
                           de l'entrée i de 1 puisque la présence
                           de LTDT a été prévue par FADA (cf. ann. 4)
                           co
            fin
   fin
```

7.4. L'ALGORITHME D'ANALYSE ASCENDANTE.

On utilise un tableau PILE comme pile d'analyse. La variable h donne la hauteur de la pile ; h est borné par hlim (nombre de cellules disponibles dans PILE). La variable B contient toujours la valeur du sommet de la pile.

```
début h:=1 ; B:=PILE[1]:=LTDT co initialisation de la pile par
                                le butoir de début co ;
     lec: LIRA(c,B,r1,r2) ;
     iter: si r2=0 alors co pas de relation de précédence entre
                            B et c co
                       ERREUR ;
     iter1: si r2=1 alors co B ≺ c co
                       début h:=h+1 ;
                             si h > hlim alors abandon du travail ;
                             B:=PILE[h]:=c ;
                             allera lec
                       fin ;
           si r2=2 alors co B ⋖ ⋗* c il faut faire une
                          réduction co
                       début iB:=B ; ih:=h ; co mémorisation de
                            l'état de la pile utile au sous-
                            programme de traitement des erreurs co
                            accès:=ENTREE[B] ;
                       essain: si accès=0 alors  allera testfin ;
                       essain1: h:=h-1 ; B:=PILE[h]co B=symbole
                                précédent sur la pile co ;
                       essai : D:=2ème demi-mot de TGA[accès]
                               co D=NDI de l'élément ou numéro
                                  de règle co ;
                               A:=1er demi-mot de TGA[accès]
                               co A=lien alternative ou NDI du
                                  membre de gauche co
                               si A ≥ 0 alors co on est en pré-
                                  sence d'un symbole de graphe
                                  susceptible de filtrer le
                                  sommet de la pile co
```

```
                début si D=B alors co le symbole
                                    D filtre co
                            début accès:=accès+1;
                                  allera essain1
                            fin ;
                      co il faut choisir une
                        autre alternative co
                      accès:=A ;
                      allera essain
                  fin ;
               co A < O, on a atteint un élément
                 FI dans le TGA co
reduc:    r2:=les 2 derniers bits de
             PRECED1[A,c] ;
          r1:=le premier bit de PRECED1[A,c] ;
          si r2=O
            alors co pas de relation entre A
                    et c ; il faut choisir
                    une autre alternative co
                  alt:
                   début accès:=TGA[accès+1]
                      co lien"alt"co ;
                      si accès=O alors ERREUR
                                 sinon allera
                                      essai
                   fin ;
          co il existe une relation de pré-
            cédence entre A et c co
          si r1=O
            alors co la réduction à faire
                    est trouvée co
            ØK: début h:=h+1 ;
                    si h>hlim
                      alors abandon du travail;
                    B:=PILE[h]:=A ;
                    sortir(D) ;
                    allera iter
                fin ;
```

```
                                co le terminal  c ne permet pas de
                                  prendre la décision ; il faut
                                  considérer le symbole B se trou-
                                  vant au sommet de la pile co
                                si ACSEC(A,B) co accès aux complé-
                                                ments des matrices de
                                                précédence co
                                   alors co on a B≺A, la réduction
                                           trouvée est la bonne co
                                         allera OK ;
                                co il faut choisir une autre alter-
                                  native co
                                  allera alt
                          fin ;
        co r2≠0, r2≠1, r2≠2 donc r2=3. Il faut réduire la
          chaîne vide co
reducvid: accès:=ENTREE[LTDT] co pour un accès aux règles
                                  vides co
          essai1: D:=2ème demi-mot de TGA[accès]co D=numéro
                                                de règle co ;
                  A:=1er demi-mot de TGA[accès] co A=NDI du
                                                membre de
                                                gauche co ;
                  allera reduc co la réduction de la chaîne
                                vide étant initialisée, elle
                                peut maintenant se poursuivre
                                comme une autre réduction co ;
testfin: co une réduction est impossible. Cet événement peut
          être provoqué, soit par une erreur dans le texte
          source,soit par la rencontre de la fin de l'analyse co
         si c ≠ LTDT alors ERREUR ;
         co le butoir de fin de chaîne est rencontré co
         si h=2 et PILE[1]=LTDT et PILE[2]=NDIAX alors allera fin;
         co le butoir est rencontré mais le contenu de la pile
          d'analyse est incorrect.Cela peut provenir soit d'une
          erreur,soit de la nécessité de réduire la chaîne vide.
         allera reducvid
fin: co de l'analyse co
       restitution de la mémoire occupée par la grammaire ;
       traitement de fin
fin
```

7.5. LE MODULE LIRA (c,B,r1,r2).

Les paramètres de ce module sont :

c = NDI du terminal lu (résultat)
B = NDI du symbole dont il doit être le voisin
r1 = si B < 0 alors 1er bit de PRECED1[B,c]
sinon 0 (résultat)
r2 = relation de précédence entre B et c (résultat)
1 si $B \lessdot c$
2 si $B \gtrdot c$ ($B \mathcal{F}^{-1+} \mathcal{F}\ \mathcal{J}^{*}\ c$)
3 si $B \gtrdot c$ ($\exists D ::= \lambda$ tel que $B \lessdot D$ et D vs c)

r2 = 0 $\Longrightarrow$ aucun terminal voisin de B n'a pu être lu ;
c et r1 n'ont alors aucune signification.

Le butoir de fin ⊢— de NDI LTDT fait partie des terminaux susceptibles d'être lus. Ce terminal a été introduit dans la table des terminaux par FADA.

Le module LIRA utilise le sous-programme LECT commun à tous les analyseurs.

```
début r1:=r2:=0 ;
      si B < 0 alors pour i:=0 pas 1 jusqua LTDT faire
                     début x:=les 2 derniers bits de PRECED1[B,i] ;
                           si x≠0 alors début LECT(i) ;
                                             si succès alors
                                               début r2:=x ;
                                                     r1:=1er bit de
                                                      PRECED1[B,i] ;
                                                     c:=i ;
                                                     RETURN
                                               fin
                                        fin
                     fin
                sinon pour i:=0 pas 1 jusqua LTDT faire
                     début x:=PRECED2[B,i] ;
```

```
                    si x≠0 alors début LECT(i) ;
                                        si succès alors
                                          début r2:=x ;
                                                c:=i ;
                                                RETURN
                                          fin
                                    fin
                        fin ;
        RETURN co retour dans le cas où aucun terminal n'a pu
                  être lu co
fin
```

7.6. LES MODULES ERREUR et ERREURR.

Le module erreur est chargé d'imprimer un message d'erreur et de reprendre l'analyse dès que possible. Il faut donc détecter un point de reprise possible.

Cette reprise peut être envisagée de deux manières différentes: ou bien la reprise est entièrement automatique, ou bien cette reprise est guidée par le programmeur FACE qui indique alors par une règle spéciale de la grammaire un ou plusieurs caractères sur lesquels peut s'effectuer la reprise.

Dans les deux cas, le premier travail des routines d'erreur est d'imprimer un message d'erreur comportant un numéro de règle erroné. Pour cela on utilise le sous-programme MESSERR (NØ) commun à tous les analyseurs. Il faut d'abord trouver le numéro de règle erroné NØ. La recherche de ce numéro est plus délicate que dans le cas de l'analyse descendante. En analyse descendante, on sait (à peu près) quelles règles sont en cours d'analyse. Ici il faut évaluer ces règles selon le contenu de la pile d'analyse.

C'est pourquoi on distingue deux points d'entrée dans la routine d'erreur : ERREUR et ERREURR.

ERREURR correspond à la détection d'une erreur durant le processus de réduction. On utilise alors la valeur de la variable accès pour parcourir le TGA jusqu'à la rencontre d'un élément FI donnant un numéro de règle; si accès = 0 on n'imprime pas de numéro (NØ = -1).

ERREUR correspond à un terminal n'ayant aucune relation de précédence avec le sommet de la pile. Il est alors fort difficile de savoir quel numéro imprimer. Dans un premier temps on n'imprimera pas de numéro dans ce cas. Au cas où le besoin s'en ferait sentir on pourrait essayer de trouver une règle en tenant compte à la fois du contenu de la pile et des relations de précédence.

```
procédure ERREUR ;
début accèsin:=accès ;
    deb: A:=1er demi-mot de TGA[accès] ;
         si A⩾0 alors début accès:=accès+1 ; allera deb fin ;
         NØ:=2ème demi-mot de TGA[accès] ;
         si NØ≠-1 alors allera notrouvé ;
         alt:=TGA[accès+1] co lien "alt" co ;
         si alt=0 alors allera remont ;
         1er demi-mot de TGA[alt-1]:=-accès co chaînage arrière co;
         accès:=alt ; allera deb ;
    remont: si accès≠accèsin  alors début accès:=accès-1 ;
                                        alt:=1er demi-mot de
                                             TGA[accès] ;
                                        si alt<0
                                          alors
                                          début 1er demi-mot de
                                                TGA[accès]:=0 ;
                                                accès:=-alt ;
                                                allera remont
                                          fin ;
                                        si alt=0
                                          alors allera remont ;
                                        co alt>0 ⟹nouvelle alter-
                                          native à explorer co
```

```
                                    ler demi-mot de
                                    TGA[alt-1]:=accès ;
                                    accès:=alt
                                    allera deb
                                fin ;
    co un numéro a été trouvé (qui peut d'ailleurs n'être que
       -1); il faut remonter jusqu'à accèsin  pour supprimer
       les liens de chaînage co
remont1: si accès≠accèsin  alors début accès:=accès-1 ;
                                    alt:=ler demi-mot de
                                         TGA[accès] ;
                                    si alt≥0 alors
                                              allera remont1 ;
                                    ler demi-mot de
                                         TGA[accès]:=0 ;
                                    accès:=alt ;
                                    allera remont1
                                fin ;
    MESSERR(NØ) ;
    co il faut maintenant effectuer la reprise de l'analyse.
       Cette partie est commune à ERREUR et ERREURR co
fin
```

La séquence correspondant à ERREUR s'écrit tout simplement :

```
ERREUR: MESSERR(-1) ; hh:=h ; allera lecc ;
```

Reprise automatique de l'analyse après une erreur.

```
        hh:=h  co valeur initiale de la hauteur de la pile co ;
exam: si B < 0 alors début r2:=les 2 derniers bits de PRECED1[B,c];
                           r1:=1er bit de PRECED1[B,c] ;
                    fin
               sinon début r2:=PRECED2[B,c] ;
                           r1:=0
                    fin ;
```

```
      si r2≠0 alors allera iter1 co la reprise est effectuée co ;
      si h > 1 alors début h:=h-1 ; B:=PILE[h] ;
                         allera exam co on essaie de trouver
                                        un point de reprise à
                                        l'intérieur de la pile co
                   fin ;
      co aucune reprise n'a été trouvée ; la pile est remise
         dans son état initial et l'on essaie de trouver un
         nouveau terminal dans le texte source co
      h:=hh ; B:=PILE[h] ;
lecc: si c=LTDT alors allera fin co fin de l'analyse co
                sinon AVANCCAR ;
      pour j:=0 pas 1 jusqua LTDT faire
      début LECT(j) ;
          si succès alors début c:=j ; allera exam fin
      fin ;
      allera lecc
```

Reprise_assistée_de_l'analyse_après_une_erreur.

La technique précédente a l'avantage d'être entièrement automatique. En revanche elle peut être assez longue et de plus la valeur des points de reprise trouvés est assez incertaine si bien qu'une seule erreur pourra parfois provoquer plusieurs messages.

Ici la reprise se fait sur un terminal particulier faisant partie d'une liste fournie par le programmeur. Lorsque l'un de ces terminaux est rencontré, on essaie de trouver une reprise possible avec un élément de la pile. Si cela s'avère impossible le terminal en question est placé à la base de la pile.

```
       pour chaque terminal de reprise j faire
       début si j=c alors allera reprise fin ;
lecc: LECT(LTDT) ; si succès alors allera fin ;
      AVANCCAR ;
      pour chaque terminal de reprise j faire
```

```
     début LECT(j) ;
           si succès alors début c:=j ; allera reprise fin
     fin ;
     allera lecc ;
reprise: si h > 1 alors début si B < 0
                              alors début r2:=2 derniers bits de
                                          PRECED1[B,c] ;
                                    r1:=1er bit de
                                          PRECED1[B,c]
                                   fin
                              sinon début r2:=PRECED1[B,c] ;
                                    r1:=0
                                   fin ;
                             si r2≠0 alors allera iter1 ;
                             h:=h-1 ;
                             B:=PILE[h] ;
                             allera reprise
                        fin ;
h:=h+1
B:=PILE[h]:=c ;
allera lec
```

Exemple d'application.

Supposons que l'on ait la règle :

<suite d'instructions>::=<suite d'instructions>;<instruction>
<instruction>

et que l'on donne comme terminal de reprise le point virgule. On peut avoir à un instant donné au sommet de la pile d'analyse, le symbole <suite d'instructions>, le symbole ";" puis une suite de symboles devant normalement se réduire en <instruction>. Si une erreur survient, on avance dans le texte source jusqu'au prochain ";" et l'on descend la pile jusqu'à ce que ce terminal puisse entrer c'est-à-dire jusqu'au non terminal <suite d'instructions>.

C H A P I T R E VIII

ANALYSEUR DESCENDANT ET ASCENDANT

La gamme de grammaires acceptées par les deux analyseurs précédents couvre une bonne partie des besoins rencontrés dans les langages de programmation habituels. Cependant les règles de déterminisme imposées, si elles ne réduisent pas l'éventail des langages acceptés, obligent parfois le programmeur FACE à quelques acrobaties dans l'écriture de la grammaire, qui ne se présente plus alors de la manière la plus naturelle.

Ce point est incontestablement une lacune puisque le but du langage FACE est l'écriture de compilateurs de la manière la plus claire et la plus naturelle possible, et de décharger cette écriture de toute préoccupation concernant l'analyse syntaxique.

Il faut noter aussi que, si lors de l'écriture de la grammaire, le programmeur peut assez facilement se rendre compte s'il respecte les règles de l'analyse descendante, il n'en est pas de même de l'analyse ascendante. En effet les règles de déterminisme (D'1) et (D'2) sont des règles locales dans la grammaire (elles portent sur les différentes réécritures d'un même non terminal ; seule la condition (D'2) demande un minimum de réflexion pour découvrir les voisins du membre de gauche); les règles (D'3) et (D'4) sont moins faciles à contrôler lors de l'écriture d'une grammaire tant soit peu importante, car elles portent sur des relations et des conditions mettant en jeu des règles de grammaire pouvant être logiquement éloignées dans l'esprit du programmeur.

Aussi le système d'analyse syntaxique présenté dans les chapitres précédents, ne prétend être qu'un début de réalisation du système d'analyse syntaxique de FACE.

Ce système doit être étendu par la réalisation de nouveaux analyseurs. Plusieurs développements sont possibles.

Il est possible de compléter les analyseurs précédents par des analyseurs indéterministes, c'est-à-dire avec retour arrière ou réalisation de plusieurs essais d'analyse en parallèle. Cependant ces solutions risquent de conduire à des

analyseurs assez lourds et à des temps d'analyse peu prévisibles pouvant être grands. Si de tels analyseurs étaient réalisés, il semble souhaitable que le préanalyseur informe alors le programmeur de la méthode d'analyse choisie, afin que celui-ci puisse s'il le désire réécrire sa grammaire afin d'obtenir un compilateur plus performant. De plus ce type d'analyseur présente une difficulté d'adaptation au langage FACE. La procédure ANAPPLYSE ne pourra plus être réalisée aussi simplement qu'elle l'est actuellement. C'est pourquoi, il semble souhaitable de se limiter, au moins dans un premier temps, à des analyseurs déterministes.

La gamme des grammaires acceptées peut aussi être étendue en élargissant le contexte permettant de prendre des décisions déterministes :

- en analyse descendante par lecture de k caractères à l'avance (grammaire LL(k)) ;

- en analyse ascendante par lecture de k caractères à l'avance et consultation de ℓ symboles au sommet de la pile ou de toute la pile, ce qui conduit aux grammaires LR(k) de KNUTH [6] ; (cet examen du contenu de la pile est facilité par le fait que les contenus de la pile conduisant à la reconnaissance d'une phrase du langage appartiennent à un langage régulier, et qu'il suffit pour tenir compte du contenu de toute la pile de se référer à l'état d'un automate fini).

Ce type de solution a certainement un intérêt pratique très important. Cependant bien que les algorithmes soient théoriquement relativement simples, ils risquent de se traduire par des programmes plus lourds que ceux réalisés actuellement et surtout par l'utilisation de tables beaucoup plus importantes, nécessitant un encombrement mémoire important (les matrices indices des matrices de relations ne sont plus des symboles mais des combinaisons de k symboles ; il est nécessaire de trouver des moyens de simplification et de représentation efficaces de ces matrices, ce qui alourdit les algorithmes).

O
O O

Il semble donc logique de rechercher un autre compromis entre l'efficacité et l'encombrement en machine d'une part et les contraintes imposées à l'utilisateur, c'est-à-dire les restrictions sur l'écriture de la grammaire d'autre part.

A première vue il semble que les contraintes imposées par les analyseurs décrits dans les chapitres précédents ne sont pas trop difficiles à respecter localement. Il peut par contre être délicat (surtout pour celles de l'analyse ascendante) de les respecter pour l'ensemble d'une grammaire qui peut être très importante. Il serait alors souhaitable d'admettre en "certains endroits" de la grammaire des infractions aux règles de déterminisme de l'analyse descendante et en d'autres endroits des infractions aux règles de déterminisme de l'analyse ascendante (cette notion d'endroit doit bien sûr être précisée). Il faut alors disposer d'un algorithme capable de passer de l'analyse descendante à l'analyse ascendante et réciproquement.

8.1. ALGORITHME D'ANALYSE ASCENDANTE _ DESCENDANTE RECURSIF ET INDETERMINISTE.

L'algorithme présenté est indéterministe. Il comporte des instructions "choix". Les instructions choix se présentent comme une suite parenthésée d'instructions du type ALGØL 60 ; les diverses instructions entre lesquelles un choix est possible sont séparées par des virgules.

Il utilise deux types d'instruction choix :

choixet : Il faut envisager tous les choix possibles. C'est le cas du choix d'une règle de la grammaire parmi toutes celles possibles. L'un au moins des choix permet d'analyser la phrase si cette dernière est correcte ; tous les choix conduisent en erreur si la phrase est incorrecte. Dans la mesure où la grammaire n'est pas ambiguë, il n'existe qu'un seul choix conduisant à une analyse correcte de la phrase.

choixou : Tous les choix conduisent à la même solution. Le choix est donc indifférent. Il faut alors trouver celui qui conduit à l'algorithme le plus simple et le plus efficace.

O
O O

L'algorithme d'analyse est composé de deux procédures :

- la procédure d'analyse descendante (analdes) ;
- la procédure d'analyse ascendante (analas).

Ces deux procédures reçoivent en paramètre un symbole qui doit être analysé. Ce symbole reçu en paramètre peut être considéré comme l'axiome d'une sous-grammaire à utiliser pour analyser une locution du texte source (la notion de sous-grammaire n'a pas actuellement de définition rigoureuse).

L'algorithme d'analyse syntaxique descendant et ascendant peut être résumé de la manière suivante :

```
début
procédure analdes(B) ;
début si B est un terminal alors lecture
                           sinon choixou analyse descendante de B,
                                         analas(B)
                                 fchoix
fin ;
procédure analas(B) ;
début tableau pa[1:n] ; entier h:=0 ;
     iter: choixou choixet lecture, réduction fchoix ,
                   essai de retour,
                   prévision d'un non terminal C à analyser
                                          et analdes(C)
           fchoix ;
           allera iter
fin ;
co programme principal : co
lire(c) ; analdes(axiome) ; si c ≠ ⊢—— alors ERREUR
fin
```

O
O O

Cet algorithme contient les procédures d'analyse syntaxique descendante et ascendante, indéterministes classiques. Des choix supplémentaires sont introduits. Il s'agit de choixou exprimant le fait que l'une ou l'autre de leurs éventualités peuvent être choisies ; chacune conduit au même résultat. Ils signifient qu'une analyse syntaxique peut être faite aussi bien en analyse descendante qu'en analyse ascendante. L'intérêt de ces choix est qu'ils autorisent, lors de la génération d'un algorithme déterministe pour une grammaire donnée, à choisir l'alternative la plus commode ou (et) la plus efficace (les choixet qui sont équivalents aux instructions choix du chapitre 2 ne sont pas résolus par le choix de la voie la plus commode, mais par la mise en oeuvre de moyens permettant de prédire à l'avance quelle est l'éventualité qui ne conduit pas à l'appel de la procédure ERREUR).

Ces choixou introduisent les possibilités suivantes :

- en analyse descendante, chaque symbole peut être analysé de façon descendante ou ascendante ; si l'analyse ascendante est choisie, le symbole à analyser sert d'axiome local à une analyse ascendante.
- en analyse ascendante :
  1) poursuivre l'analyse ascendante (lecture ou réduction)
  2) rendre le contrôle (si une locution correcte a été analysée)
  3) faire une prévision sur le prochain non terminal à analyser et l'analyser en analyse descendante.

O
O O

L'algorithme suivant utilise une grammaire (G=X,::=,N,T) composée de règles :

$A_k ::= B_{k1} \ldots B_{kq_k}$ (règle n° k)

La pile utilisée en analyse ascendante est notée pa ; sa hauteur (nombre de symboles qu'elle contient) est notée h. pa $\setminus \lambda$ désigne le contenu de la pile pa amputé de son facteur droit λ. Cette opération ainsi que la concaténation à pa (notée sans signe) a pour effet de bord, la mise à jour de la hauteur h.

$\jmath$ est une fonction fournissant le numéro d'une règle :

$\jmath\ (A_k ::= B_{k1} \ \ldots \ B_{kq_k}) = k$

```
début
procédure analdes(B) ;
début si B ∈ T
      alors co B terminal co
            début si B = c co c = terminal lu à l'avance co
                  alors début sortir(c) ;
                              lire(c)
                        fin
                  sinon erreur
            fin
      sinon début co analyse descendante indéterministe classique:co
                  choixet choix d'un numéro de règle k tel que A_k=B
                  fchoix ;
                  pour j:=1 pas 1 jusqua q_k faire
                  co choix entre l'analyse descendante et l'analyse
                     ascendante, choix fait pour chaque symbole du
                     membre de droite de la règle co
                  choixou analdes(B_kj) ,
                          analas(B_kj)
                  fchoix ;
                  sortir(k)
            fin
fin ;
procédure analas(B) ;
début tableau pa[1:n] co pile d'analyse ascendante co ;
      entier h:=0 co la hauteur de la pile pa co ;
      co ces déclarations sont des déclarations locales. Il est
         fondamental de déclarer ces variables locales car la
         procédure analas est récursive, non directement, mais
         par l'appel de analdes lorsque l'on fait une prévision co ;
```

```
iter: choixou co 1) analyse ascendante normale co
                choixet co lecture co
                        si c≠⊢— alors début h:=h+1 ;
                                           pa[h]:=c ;
                                           sortir(c) ; lire(c)
                                    fin
                                sinon si h=1 et pa[1]=B
                                        alors return
                                        sinon ERREUR,
                        co réduction co
                        début choixet de λ au sommet de pa et
                                      de A tel que A::=λ et
                                      A vs c
                              fchoix ;
                              pa:=(pa \ λ) A ;
                              sortir( ) (A::=λ))
                        fin,
                co essai de retour co
                si h=1 et pa[1]=B alors return,
                co prévision co
                début C:=pa[h] ;
                      choixet de A tel que (si h=0 alors B  +  A
                                            sinon C  *  A)
                                        et A  *  c fchoix ;
                      analdes(A) ;
                      h:=h+1 ; pa[h]:=A
                fin
        fchoix ;
        allera iter
fin ;
co programme principal : co
lire(c) ; analdes(X) ;
si c ≠ ⊢—— alors ERREUR
fin
```

o
o o

La séparation de l'analyse ascendante et descendante en deux procédures distinctes n'est faite que dans un but de clarté d'écriture, mais n'est nullement nécessaire à l'expression de l'algorithme. Il serait possible d'écrire une seule procédure récursive en remplaçant dans la procédure analdes, l'appel de la procédure analas par son expression.

Il faut cependant noter que la procédure analas est elle aussi récursive puisqu'elle appelle la procédure analdes qui peut la rappeler. C'est pourquoi il est important que la pile d'analyse ascendante soit réservée localement à chaque appel afin de permettre la récursivité. Dans la pratique les différentes piles d'analyse ascendante et descendante (servant à résoudre la récursivité) pourront être représentées dans une même pile en mémoire à l'aide de marqueurs de pile. Cela résulte de ce que les piles sont utilisées selon une structure de pile : à un moment donné on n'accède jamais qu'au sommet de la dernière pile créée.

8.2. EXEMPLE D'UTILISATION DE L'ALGORITHME.

Soit la grammaire suivante d'axiome <suite d'instructions>:

<suite d'instructions>::=<instruction> | <suite d'instructions> ; <instruction>
<instruction>::=<affectation> | <instr. cond.>
<affectation>::=<ident > = <expr. arith.>
<instr. cond.>::=si <expr. bool.> alors <suite d'instructions> <fin instr. si>
<fin instr. si>::=fsi | sinon <suite d'instructions> | fsi
<expr. bool.>::=<terme bool.> | <expr. bool.> + <terme bool.>
<terme bool.>::=<fact. bool.> | <terme bool.> * <fact. bool.>
<fact. bool.>::=<prim. bool.> | non <prim. bool.>
<prim. bool.>::=<relation> | (<expr. bool.>)
<relation>::=(<expr. arith.> = <expr. arith.>)
<expr. arith.>::=<terme> | <expr. arith.> + <terme>
<terme>::=<facteur> | <terme> * <facteur>

<facteur>::=<symbole> | <nbre> | (<expr. arith.>) | <expr. cond.>
<expr. cond.>::=si <expr. bool.> alors <expr. arith.> sinon
<expr. arith.> fsi
<symbole>::=<ident> (<indice>) | <ident>
<indice>::=<ident> | <nbre>

O
O O

Cette grammaire ne peut être analysée avec l'analyseur descendant du chapitre 5 car les deux règles de réécriture de <prim. bool.> ont une initiale terminale commune "(". Cette indétermination ne serait pas levée non plus par un analyseur descendant lisant un nombre fixe de caractères à l'avance (grammaire non LL(k)).

Elle ne peut pas non plus être analysée avec l'analyseur ascendant du chapitre 7. Il existe en effet des conflits de précédence entre :

<ident> et le signe =
<expr. arith.> et le symbole sinon
<expr. arith.> et le symbole fsi

De plus <indice> et <facteur> ont des parties droites identiques et (≺ <facteur>
(≺ <indice>
<facteur> vs)
<indice> vs).

La sous-grammaire d'axiome <expr. bool.> ne contient pas de conflit de précédence.

Si dans cette sous-grammaire, on considère indice comme un terminal, elle peut faire l'objet d'une analyse ascendante.

Si dans le reste de la grammaire on ne se préoccupe pas des expressions booléennes, une analyse descendante est possible.

o
o o

Un analyseur pour cette grammaire s'écrit :

```
début
procédure suite d'instructions ;
début instruction ; sortir (ν)
     deb: si c=";" alors début lire(c) ; instruction ; sortir(ν) ;
                               allera deb
                          fin
fin ;
procédure instruction ;
si c="si" alors début instr. cond. ; sortir(ν) fin
          sinon début affectation ; sortir(ν) fin ;
procédure affectation ;
début si c≠ident alors ERREUR ;
      lire(c) ;
      expr. arith. ;
      sortir(ν)
fin ;
procédure expr. arith. ;
début terme ; sortir(ν) ;
      deb: si c="+" alors début lire(c) ; terme ; sortir(ν) ;
                               allera deb
                          fin
procédure terme ;
début facteur ; sortir(ν) ;
     deb: si c="*" alors début lire(c) ; facteur ; sortir(ν) ;
                               allera deb
                          fin
procédure facteur ;
début si c="ident" alors début symbole ; sortir(ν) ;
                               return
                          fin ;
      si c="nbre" alors début lire(c) ; sortir(ν) ;
                               return
                          fin ;
```

```
        si c="(" alors début lire(c) ;
                              expr. arith. ;
                              si c≠")" alors ERREUR ;
                              sortir(ν) ; return
                         fin ;
        expr. cond. ; sortir(ν)
fin ;
procédure expr. cond. ;
début si c≠"si" alors ERREUR ;
        lire(c) ;
        expr. bool. ;
        si c≠"alors" alors ERREUR ;
        lire(c) ;
        expr. arith. ;
        si c≠"sinon" alors ERREUR ;
        lire(c) ;
        expr. arith. ;
        si c≠"fsi" alors ERREUR ;
        lire(c) ; sortir(ν)
fin
procédure symbole ;
début si c≠"ident" alors ERREUR ;
        lire(c) ;
        si c="(" alors début lire(c) ; indice ;
                              si c≠")" alors ERREUR ;
                              lire(c) ;
                              sortir(ν)
                         fin
                   sinon sortir(ν)
 fin ;
 procédure instr. cond. ;
 début si c≠"si" alors ERREUR ;
        lire(c) ;
        expr. bool. ;
        si c≠"alors" alors ERREUR ;
        lire(c) ;
        suite d'instructions ;
        fin  instr. si ; sortir(ν)
 fin ;
```

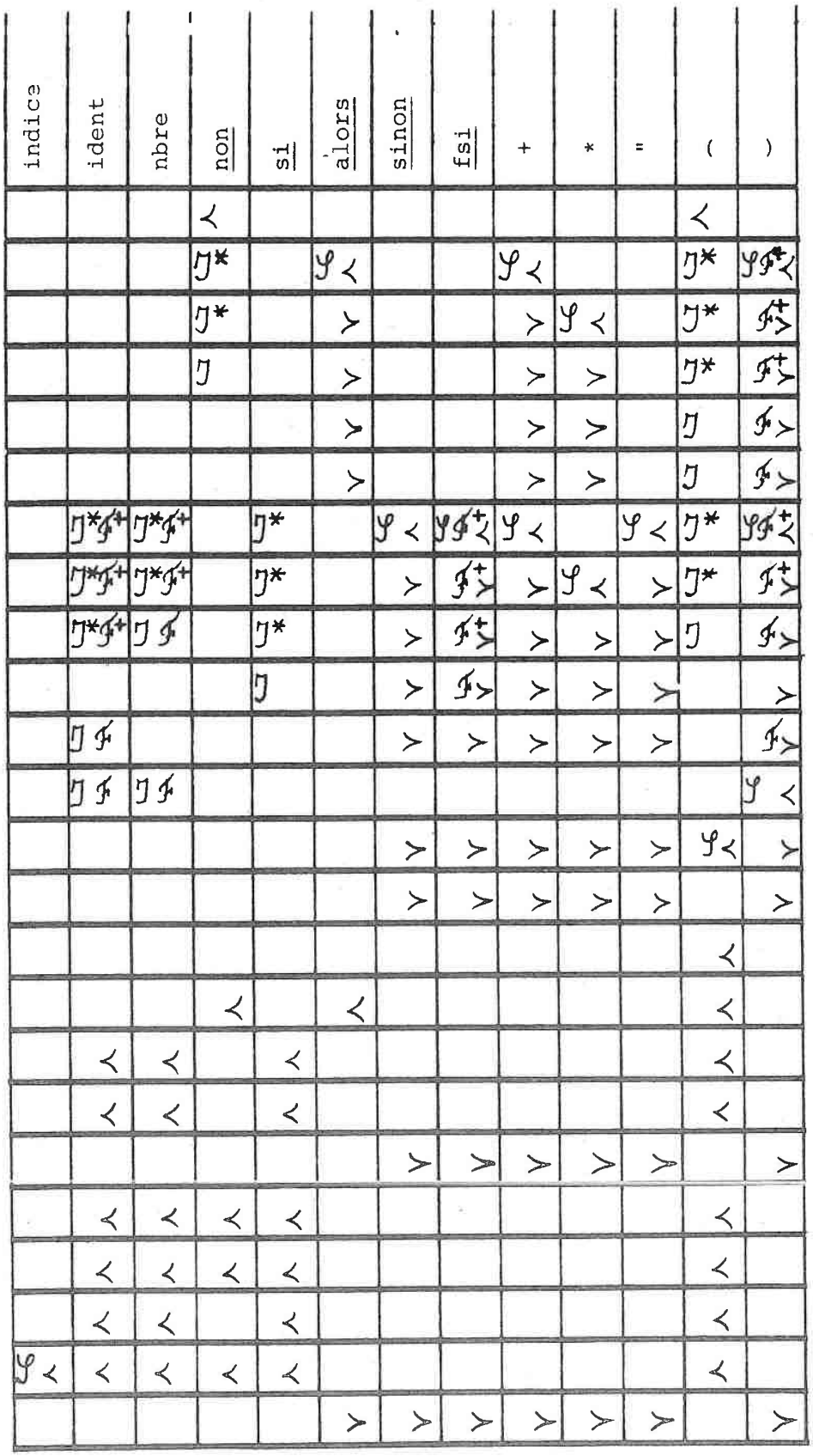
```
procédure fin instr. si ;
 si c="fsi" alors début sortir(ν) ; lire(c) fin
            sinon début si c≠"sinon" alors ERREUR ;
                        lire(c) ;
                        suite d'instr. ;
                        si c≠"fsi" alors ERREUR ;
                        lire(c) ; sortir(ν)
                  fin ;
procédure expr. bool. ;
début tableau pa[1:n] ; entier h:=1 ; B:=pa[h]:= ——┤ ;
      co ——┤ est le butoir de début, on a ——┤ ℑ ℐ * ( et
         ——┤ ℑ ℐ * non co
 iter: si c="alors" et pa=( ——┤ , <expr. bool.>) alors return ;
       si B="ident" et c="(" alors début empiler(c) ;
                                        lire(c) ;
                                        indice ;
                                        empiler(indice) ;
                                        allera iter
                                   fin
       si B ≺ c alors début empiler(c) ; lire(c) ; allera iter fin
       si non B ≻ c alors ERREUR ;
       réduction ; sortir(ν)
       allera iter
fin

procédure indice ;
si c="ident" alors début lire(c) ; sortir(ν) fin
             sinon début si c≠"nbre" alors ERREUR ;
                         lire(c) ; sortir(ν)
                   fin
```

VIII.13 Les relations de précédence utilisées dans la procédure expr. bool. sont :

	expr. bool.	terme bool.	fact. bool.	prim. bool.	relation	expr. arith.	terme	facteur	expr. cond.	symbole
⊢	≐ ≺	≺	≺	≺	≺					
expr. bool.	≐	≐ ≑	≐*≑+	≐*≑+	≐*≑+					
terme bool.		≐	≐ ≑	≐*≑+	≐*≑+					
fact. bool.				≐ ≑	≐*≑+					
prim. bool.					≐ ≑					
relation										
expr. arith.						≐	≐ ≑	≑+	≐*≑+	≐*≑+
terme								≑	≐*≑+	≐*≑+
facteur									≐ ≑	≐ ≑
expr. cond.										
symbole										
indice										
ident										
nbre										
non				≐ ≺						
si	≐ ≺	≺	≺	≺	≺					
alors						≐ ≺	≺		≺	≺
sinon						≐ ≺	≺		≺	≺
fsi										
+		≐ ≺	≺	≺	≺		≐ ≺	≺	≺	≺
*			≐ ≺	≺	≺			≐ ≺	≺	≺
=						≐ ≺	≺	≺	≺	≺
(	≐ ≺	≺	≺	≺	≺	≐ ≺	≺	≺	≺	≺
)										

Ce tableau montre que la sous-grammaire des expressions booleennes ne contient pas de conflit de précédence et



peut être utilisée pour une analyse descendante.

PROLONGEMENTS POSSIBLES

Le chapitre 8 ne contient que l'ébauche d'un analyseur descendant-ascendant. Il reste à étudier plus en détail l'arrêt de l'analyse ascendante. L'analyse ascendante peut, soit s'arrêter lorsqu'une locution dérivant de son paramètre est analysée, soit se poursuivre plus loin ; dans ce cas la pile d'analyse ascendante ne serait pas vide au moment du retour ; elle devrait être rendue à l'analyse descendante appelante pour lui servir de ruban d'entrée.

Pour utiliser cet algorithme en FACE, il est nécessaire de réaliser la génération automatique d'un tel analyseur à partir d'une grammaire. Cela implique la définition de la notion de sous-grammaire et l'étude de la partition d'une grammaire en sous-grammaires en fonction des conflits aux règles de déterminisme (D'1), (D'2), (D3), (D4) et (D5).

Il serait intéressant de pouvoir définir exactement la classe de grammaires acceptées par un tel algorithme et de la comparer aux grammaires acceptées par les algorithmes des chapitres 5 et 7 d'une part et aux grammaires LR(k) d'autre part.

o
o o

Un autre prolongement du travail qui vient d'être présenté consiste en des aides à la mise au point de compilateurs plus efficaces que ceux dont on dispose actuellement. A cette fin le préanalyseur FADA, une fois rendu conversationnel, devrait permettre l'écriture des grammaires de façon interactive. Un tel préanalyseur informerait le programmeur de l'efficacité de l'analyseur qui sera utilisé, en fonction de la complexité de la grammaire qu'il propose. Il serait intéressant aussi de pouvoir définir une grammaire, non plus globalement, mais par blocs logiques indépendants (grammaire des expressions, grammaire des déclarations, grammaire des instructions conditionnelles...) mis au point et testés séparément. Le préanalyseur conversationnel devrait mémoriser ces différents blocs logiques et les assembler automatiquement pour constituer la grammaire résultante puis son analyseur syntaxique. Il semble intuitive-

ment que cette notion de bloc logique doit correspondre à celle de sous-grammaire de l'analyseur descendant-ascendant. Il n'y a donc pas obligation à ce que le découpage en sous-grammaires soit toujours réalisé automatiquement. Le découpage manuel, outre l'avantage de simplicité de réalisation qu'il présente, peut permettre à l'utilisateur un plus grand contrôle des passages d'un type d'analyse à l'autre, ces passages pouvant éventuellement avoir une incidence sur l'encombrement mémoire et l'efficacité de l'analyseur obtenu.

A N N E X E 1

DESCRIPTION DES PROCEDURES :

CALL

SAVE

RETURN

RESTAUR

LA PROCEDURE *CALL*

La procédure CALL permet d'appeler un sous-programme en plaçant éventuellement des paramètres dans la séquence d'appel.

CALL peut être codé

[étiquette1][,étiquette2] *CALL*,CF [ARG[,ARG]*]

CF s'écrit *spg* ou *(spg,n)* *n* doit être un numéro de registre d'index
ARG s'écrit [*]*symb*

1) Si CF est écrit *spg* CALL effectue un appel du sous-programme d'adresse spg, par l'intermédiaire du registre de liaison @RL, c'est-à-dire qu'il génère une instruction BAL,@RL spg.

2) Si CF est écrit *(spg,n)* le branchement est effectué à l'adresse spg indexée du contenu du registre n, c'est-à-dire que l'instruction générée est : BAL,@RL spg,n.

3) A la suite de l'instruction BAL, chaque *symb* figurant en argument, est mémorisé sur un mot dans l'ordre où il figure dans la ligne de référence.

4) Les arguments codés **symb* désignent des paramètres se trouvant placés en mémoire à l'adresse *symb*. Pour ces arguments CALL génère avant l'instruction BAL, les instructions nécessaires pour placer ces paramètres dans la séquence d'appel.

5)*étiquette1* repère la première instruction générée par CALL.

6)*étiquette2* repère l'instruction BAL.

Exemples de codes générés : @RL étant le registre 2

```
          CALL,SPG                    génère              BAL,2    SPG
ETI1,ETI2 CALL,SPG                    génère    ETI1,ETI2 BAL,2    SPG
ETI1      CALL,(SPG,4)                génère    ETI1      BAL,2    SPG,4
ETI1      CALL,SPG      A,B,4         génère    ETI1      BAL,2    SPG
                                                          DATA,4   A
                                                          DATA,4   B
                                                          DATA,4   4
ETI1,ETI2 CALL,(SPG,5)  A,*B,4,C,*D   génère    ETI1      LW,2     B
                                                          STW,2    ETI2 + 2
                                                          LW,2     D
                                                          STW,2    ETI2 + 5
                                                ETI2      BAL,2    SPG,5
                                                          DATA,4   A
                                                          DATA,4   B
                                                          DATA,4   4
                                                          DATA,4   C
                                                          DATA,4   D
```

LA PROCEDURE *SAVE*

La procédure SAVE est normalement utilisée à l'entrée d'un programme pour :

- réserver éventuellement une zone de sauvegarde des registres à l'usage des sous-programmes pouvant être appelés par le programme en cours ;

- effectuer la sauvegarde des (ou de certains) registres dans la zone de sauvegarde du programme appelant.

Lorsque SAVE est codé un registre (@ASM, généralement le registre 1) doit contenir l'adresse de la zone de sauvegarde du programme appelant, à moins qu'il n'y ait aucun registre à sauvegarder, ce qui est possible.

SAVE peut être codé

[*étiquette*] *SAVE*[,CF] [ARG[,ARG[,ARG]]]

ARG s'écrit ARG1 ou ARG2 ou *NØL*

Si plusieurs arguments sont présents (2 ou 3) on ne doit jamais avoir deux arguments de la même forme.

ARG1 s'écrit *(SAVMEM[,symb])*
ou *SAVMEM*

ARG2 s'écrit $(m_1[,m_2])$
ou m_1

CF s'écrit $(n_1[,n_2])$
n_1

m_1, m_2, n_1, n_2 ont une valeur comprise entre 0 et 15 et peuvent être n'importe quelle expression METASYMBOL calculable.

Les arguments ARG1 et ARG2 servent à indiquer la manière dont doit être faite la réservation :

1) En l'absence de tout argument une zone standard de 16 mots est réservée.

2) Si ARG1 est présent et que *symb* est omis (c'est-à-dire : *(SAVMEM)* ou *SAVMEM*) et quel que soit ARG2, aucune zone n'est réservée.

3) Si SAVMEM est accompagné d'un symbole (*symb*) ce symbole identifie la zone réservée.

4) Si l'on est dans le cas (2), ARG2 peut être codé mais n'a aucune signification. ARG2 sert à préciser la taille de la zone à réserver. En l'absence de ARG2 la taille standard réservée est de 16 mots.

5) Si ARG2 est présent, mais que m_2 est omis ((m_1) ou m_1) la zone réservée est de (m_1 + 1) mots, taille nécessaire et suffisante pour sauvegarder m_1 registres.

6) Si m_2 est présent, une zone est réservée, nécessaire et suffisante pour préserver les registres m_1 à m_2 modulo 16.

Lorsque SAVE effectue une réservation, il ouvre une nouvelle section dans laquelle il fait cette réservation, puis revient dans la section courante.

CF sert à indiquer les registres à sauvegarder.

7) Si CF est omis, 15 registres sont sauvegardés, les registres @ASM + 1 à @ASM - 1 (modulo 16).

8) Si CF est présent et que n_2 est omis, les registres sauvegardés sont @ ASM + 1 à @ASM + n_1 modulo 16.

9) Si n_1 et n_2 sont présents, les registres sauvegardés sont les registres n_1 à n_2 modulo 16.

La sauvegarde commence toujours au mot 1 (2ème mot) de la zone dont l'adresse est contenue dans @ASM.

Si une (ou plusieurs) étiquette est présente dans la ligne de référence, elle identifie la première instruction générée sur SAVE dans la section courante.

Exemples de codes générés : @ASM étant le registre 1

```
ETI   SAVE                                    16 mots sont réservés dans
                                              une autre section
                                        ETI   STW,1     ad   zone réservée
                                              LCI       15
                                              STM,2     1,1
                                              LI,1      ad   zone réservée

ETI   SAVE          SAVMEM              ETI   LCI       15
                                              STM,2     1,1

      SAVE,(12,0)  (SAVMEM,TRUC),(2,5)        5 mots réservés à l'adresse
                                              TRUC
                                              STW,1     TRUC
                                              LCI       5
                                              STM,12    1,1
                                              LI,1      TRUC
      SAVE,3       (SAVMEM,TRUC),5,NØL        6 mots réservés à l'adresse
                                              TRUC
                                              STW,1     TRUC
                                              LCI       3
                                              STM,2     1,1

            @ ASM étant le registre 9
ETI   SAVE,(14,4) (SAVMEM,TRUC),2             3 mots réservés à l'adresse
                                              TRUC
                                        ETI   STW,9     TRUC
                                              AI,9      1
                                              LCI       7
                                              STM,14    *9
                                              LI,9      TRUC

      SAVE,(14,14) (SAVMEM,TRUC),(13,2)       7 mots réservés à l'adresse
                                              TRUC
                                              AI,9      1
                                              STW,14    *9
                                              LI,9      TRUC
```

On constate que dans chaque cas, le code généré est le plus efficace possible.

Ces exemples montrent aussi que lorsqu'une zone est réservée, la valeur du registre @ASM est rangée dans le premier mot de cette zone, et que, après sauvegarde des registres, l'adresse de cette zone est chargée dans @ASM. Cette dernière action peut cependant être inhibée par la présence du paramètre NØL dans la ligne de référence.

SAVE peut aussi réserver une zone en ne sauvegardant aucun registre : ETI SAVE,0 génère la réservation de 16 mots
et ETI STW,@ASM adresse zone réservée.

Si SAVE ne demande pas de réservation, on peut signaler l'existence d'une zone de sauvegarde existante afin qu'il gère @ASM : *(SAVEXIST,adresse)* option qui doit figurer avec : *SAVMEM* ou *(SAVMEM)* ou *(SAVMEM,)*.

LA PROCEDURE *RETURN*

La procédure RETURN peut être utilisée dans un programme pour rendre le contrôle au programme appelant. RETURN effectue :

- la restauration des registres,
- le retour au programme appelant.

RETURN peut être codé

[*étiquette*] *RETURN*[,CF] [ARG[,ARG[,ARG[,ARG]]]]

ARG s'écrit ARG1 ou ARG2 ou ARG3 ou ARG4. Si plusieurs arguments sont présents (2, 3 ou 4) on ne doit jamais trouver deux arguments de la même forme.

ARG1 s'écrit *(SAVMEM*[,SARG]*)* ou *SAVMEM*

SARG s'écrit [*]*symb* ou *NØL*

ARG2 s'écrit *(DEP,m)*

ARG3 s'écrit *(MARS,ℓ)*

ARG4 s'écrit *(PARM,n)*

CF s'écrit *(*n_1[,n_2]*)* ou n_1

m, *ℓ*, n_1 et n_2 doivent être des expressions METASYMBOL correctes et avoir une valeur comprise entre 0 et 15. *n* est une expression METASYMBOL pouvant avoir n'importe quelle valeur entière ≥ 0.

L'*étiquette* pouvant figurer devant *RETURN* identifie la première instruction générée par RETURN.

Avant que RETURN puisse faire la restauration des registres l'adresse de la zone de sauvegarde doit être chargée dans le registre @ASM. ARG1 permet de définir l'adresse de la zone de sauvegarde :

1) Si ARG1 est absent, RETURN prend une position en corrélation avec le dernier SAVE rencontré : si le dernier SAVE a modifié le registre @ASM, RETURN produit un ordre LW,@ASM *@ASM ou LW,@ASM o,@ASM (si @ASM est un registre d'index), et considère qu'après exé-

cution de cet ordre @ASM contient l'adresse de la zone de sauvegarde ; si le dernier SAVE n'a pas modifié @ASM, RETURN ne le modifie pas non plus, et considère qu'au moment de l'appel, @ASM contient l'adresse de la zone de sauvegarde.

2) Si ARG1 est codé *(SAVMEM)* ou *SAVMEM*, RETURN produit un ordre LW, @ASM *@ASM ou LW, @ASM 0, @ASM, et considère alors que la bonne adresse est chargée dans @ASM.

3) Si *symb* ou **symb* est présent dans ARG1, RETURN produit un ordre LW, @ASM [*]symb ou même LW, @ASM 0, symb (si symb est un numéro de registre d'index), et considère l'adresse de la zone de sauvegarde chargée dans @ASM.

4) Si *NØL* est codé dans ARG1, RETURN ne modifie pas @ASM.

CF définit les registres à restaurer :

5) Si CF est absent les registres restaurés sont les mêmes que ceux qui ont été sauvegardés par le dernier SAVE rencontré.

6) Si CF est présent mais que n_2 est omis, n_1 registres sont restaurés, le premier étant le premier registre sauvegardé par le dernier SAVE rencontré (si n_1 = 0 aucune restauration n'est faite).

7) Si n_1 et n_2 sont présents les registres portant les numéros n_1 à n_2 (modulo 16) sont restaurés.

8) La restauration des registres se fait normalement en accord avec la dernière instruction SAVE rencontrée, c'est-à-dire que dans le cas où le premier registre restauré, IRR, est différent du premier registre sauvegardé, RETURN restaure les registres à partir d'une adresse relative à la zone de sauvegarde identique à celle où IRR a été rangé par SAVE.

9) L'adresse relative dans la zone de sauvegarde, à partir de laquelle est restauré le premier registre peut être imposée par le programmeur en codant ARG2: *(DEP,m)* ; m doit avoir une valeur comprise entre 0 et 14. La valeur 0 fait que le premier registre à restaurer est chargé du contenu du 2ème mot de la zone de sauvegarde. Il est possible que le déplacement amène à charger dans un registre le dernier mot de la zone de sauvegarde alors qu'il reste des registres à restaurer. Alors, si la zone de sauvegarde est

de 16 mots, les registres suivants sont restaurés à partir du 2ème mot de la zone de sauvegarde.

10) Dans le cas où la zone de sauvegarde est plus petite que 16 mots, le fait de restaurer des registres différents de ceux sauvegardés par le dernier SAVE, ou d'imposer un déplacement *(DEP,m)* est dangereux car cela peut amener la restauration de certains registres à partir de mot mémoire n'appartenant pas à la zone de sauvegarde. Pour y remédier, il est fortement conseillé de passer dans ces deux cas, la longueur de la zone de sauvegarde à RETURN. Ceci se fait par l'intermédiaire de ARG3 : *(MARS,ℓ)* où ℓ est le nombre maximum de registres que l'on peut sauvegarder dans la zone considérée. En l'absence de ARG3, la valeur implicite de ℓ est 15 (cf (9)).

11) Après avoir éventuellement : chargé le registre @ASM puis effectué la restauration des registres, RETURN effectue le retour au programme appelant. L'adresse de retour doit se trouver dans le registre de liaison @RL.

12) Le retour peut cependant être effectué à une adresse supérieure à la valeur contenue dans @RL, ceci pour tenir compte en particulier des paramètres pouvant être placés dans la séquence d'appel. Il faut alors faire figurer ARG4 dans la ligne de référence : *(PARM,n)* où *n* est l'adresse de retour relative par rapport au contenu de @RL (*n* est généralement égal au nombre de paramètres passés dans la séquence d'appel).

Exemples de codes générés : @ASM et @RL étant les registres 1 et 2

```
SAVE
RETURN                                          LW,1    O,1
                                                LCI     15
                                                LM,2    1,1
                                                B       O,2

SAVE            SAVMEM
RETURN,(5,9)    (PARM,3)                        LCI     5
                                                LM,5    4,1
                                                B       3,2
RETURN,(5,9)    (SAVMEM,NØL),(DEP,14)           LW,5    15,1
                                                LCI     4
                                                LM,6    1,1
                                                B       O,2
RETURN,(5,9)    (SAVMEM,TRUC),(DEP,7),(MARS,9)  LW,1    TRUC
                                                LCI     2
                                                LM,5    8,1
                                                LCI     3
                                                LM,7    1,1
                                                B       O,2
```

LA PROCEDURE *RESTAUR*

La procédure RESTAUR est utilisée pour restaurer les (ou certains) registres.

RESTAUR peut être codé:

[*étiquette*] *RESTAUR*[,CF] [ARG[,ARG[,ARG]]]

ARG s'écrit ARG1 ou ARG2 ou ARG3

Si plusieurs arguments sont présents (2 ou 3) on ne doit jamais en trouver deux de la même forme.

ARG1, ARG2, ARG3 et CF se codent exactement comme les paramètres correspondants de la procédure RETURN et ont la même signification.

L'*étiquette* pouvant figurer devant *RESTAUR* identifie la première instruction générée par RESTAUR.

RESTAUR effectue la restauration des registres exactement comme le fait RETURN, et CF, ARG1, ARG2, ARG3 influent sur cette restauration et sur le chargement éventuel de ASM exactement de la même manière que dans RETURN.

Les deux instructions : { RESTAUR / RETURN,(0) (SAVMEM,NOL) } ont un effet strictement identique à l'instruction : RETURN.

La procédure RESTAUR peut être utile dans l'écriture d'un sous-programme à plusieurs entrées. Soit par exemple un sous-programme à deux entrées E1 et E2, et que l'entrée E1 consiste à dérouler une séquence particulière avant d'entrer dans la séquence commune débutant en E2, on pourra écrire :

```
E1   SAVE      (SAVMEM,SAVEA)
     |
     RESTAUR
E2   SAVE      (SAVMEM),(SAVEXIST,SAVEA)
     |
     RETURN
```

LA PROCEDURE *RESTAUR*

La procédure RESTAUR est utilisée pour restaurer les (ou certains) registres.

RESTAUR peut être codé:

[*étiquette*] *RESTAUR*[,CF] [ARG[,ARG[,ARG]]]

ARG s'écrit ARG1 ou ARG2 ou ARG3

Si plusieurs arguments sont présents (2 ou 3) on ne doit jamais en trouver deux de la même forme.

ARG1, ARG2, ARG3 et CF se codent exactement comme les paramètres correspondants de la procédure RETURN et ont la même signification.

L'*étiquette* pouvant figurer devant *RESTAUR* identifie la première instruction générée par RESTAUR.

RESTAUR effectue la restauration des registres exactement comme le fait RETURN, et CF, ARG1, ARG2, ARG3 influent sur cette restauration et sur le chargement éventuel de ASM exactement de la même manière que dans RETURN.

Les deux instructions : { RESTAUR / RETURN,(0) (SAVMEM,NOL) } ont un effet strictement identique à l'instruction : RETURN.

La procédure RESTAUR peut être utile dans l'écriture d'un sous-programme à plusieurs entrées. Soit par exemple un sous-programme à deux entrées E1 et E2, et que l'entrée E1 consiste à dérouler une séquence particulière avant d'entrer dans la séquence commune débutant en E2, on pourra écrire :

```
E1    SAVE      (SAVMEM,SAVEA)
      |
      RESTAUR
E2    SAVE      (SAVMEM),(SAVEXIST,SAVEA)
      |
      RETURN
```

A N N E X E 2

LES SOUS-PROGRAMMES :

LECT(NDI)

MESSERR(NO)

AVANCCAR

- Reconnaissance d'un terminal
- Impression des messages d'erreur
- Lecture du texte source
- Edition d'un listing source
- Avancement d'un caractère dans le texte source

Ces sous-programmes sont communs à l'analyseur descendant et ascendant.

LE MODULE LECT(NDI)

Module commun à l'analyseur descendant et ascendant, utilisé pour effectuer la lecture et le listing du texte source et reconnaître l'occurence d'un symbole terminal.

Paramètre : NDI = n° du terminal à lire.

Résultat : succès : le terminal a été reconnu comme étant celui qui se présentait dans le texte source.

échec : le terminal n'a pas été reconnu. Dans ce cas, le texte source reste intact devant la fenêtre : le ruban d'entrée n'avance pas.

Principe :

En fait, le sous-programme LECT se contente d'isoler un à un chaque caractère du texte source et il les confie à un sous-programme de reconnaissance spécialisé (TEST).

Ce sous-programme spécialisé peut être soit un sous-programme standard de reconnaissance d'un terminal soit un sous-programme fourni par l'utilisateur.

Il doit rendre à LECT un code de retour pouvant prendre l'une des 3 valeurs suivantes :

- succès (le terminal est reconnu)
- échec (reconnaissance impossible)
- en cours (le caractère suivant doit être examiné)

Le caractère confié au sous-programme de reconnaissance ne peut pas être un caractère à sauter ; ces caractères sont éliminés par le sous-programme LECT qui s'occupe de la gestion de ces caractères et du saut de ligne.

Technique de bufferisation :

Le sous-programme LECT doit résoudre certains problèmes pratiques de présentation du texte source.

On suppose que le texte source est présenté en enregistrements de LENR octets (en général LENR = 80).

Dans le cas où l'enregistrement doit être listé, les 80 caractères sont listés (accompagnés d'un numéro de ligne).

Cependant sur l'enregistrement les premiers octets ou les derniers peuvent être réservés pour des commentaires, une identification du programme ou un numéro de séquence.

Un certain nombre d'octets NDEB peuvent être réservés au début de l'enregistrement. On trouve ensuite le texte source sur LENRU octets.

Les symboles LENR, NDEB et LENRU sont en fait des constantes. Elles sont définies comme des paramètres de génération de l'analyseur. Elles peuvent être modifiées mais cela nécessite un réassemblage et une réédition de l'analyseur. On aura par exemple :

```
LENR    EQU   80
NDEB    EQU   0
LENRU   EQU   72
```

Les enregistrements correspondent à un découpage physique du texte source et non à un découpage logique.

Les terminaux qui constituent les unités logiques de ce texte peuvent avoir une longueur quelconque, être entrecoupés de caractères à sauter, s'étendre sur plusieurs enregistrements physiques.

Les problèmes ainsi soulevés sont résolus par LECT de la manière suivante :

On utilise deux buffers de lecture B1 et B2 comportant chacun LENR octets. La partie utile de ces buffers (chacun LENRU octets) commence respectivement en BUF1 et BUF2. On a :

```
BUF1    EQU    B1 + NDEB/4
BUF2    EQU    B2 + NDEB/4
```

Dans le cas d'une densité trop importante de caractères à sauter ces deux buffers peuvent être insuffisants. On dispose alors d'une zone RESERVE de taille égale à 2 × LENRU, dans laquelle on peut emmagasiner le texte source dépouillé des caractères à sauter. Cette méthode permet au niveau de l'analyse lexicographique des allers et retours d'amplitude importante dans le texte et indépendants de la mise en page choisie par le programmeur à l'aide des caractères à sauter. (Un seul inconvénient est rencontré : une détection d'erreur lorsqu'on travaille sur la réserve se fait alors que la ligne a déjà été imprimée sur le listing et que d'autres l'ont été depuis ; un tel événement doit cependant se produire très rarement).

Problème des caractères à sauter :

Chaque caractère à sauter peut entraîner la sortie d'un numéro de règle ; cette sortie ne doit pas se faire immédiatement mais doit être différée jusqu'à ce que l'on avance dans la grammaire sur un lien"suc".Aussi les numéros de règle seront empilés et sortis par l'analyseur au bon moment.

Le saut de ligne entre dans ce cadre.

Les caractères à sauter peuvent être rencontrés n'importe où ; y compris à l'intérieur d'un terminal.

Au cours de LECT les caractères à sauter sont empilés dans la pile PILSAU. Cette pile est détruite en cas d'échec, conservée en cas de succès. (Ainsi un caractère à sauter n'est bien compté qu'une fois).

Analyse détaillée :

Le sous-programme LECT utilise 2 variables rémanentes pour déterminer le point où l'on en est dans le texte source (la fenêtre).

Ces variables sont B et I.
B indique la zone de mémoire en cours d'utilisation :

B = 1 pour BUF1
B = 2 pour BUF2
B = 3 pour RESERVE.

I indique l'indice du caractère à examiner dans la zone considérée.

Au moment de l'appel de LECT, les variables internes BC et IC prennent les valeurs de B et C (respectivement). BC et IC sont utilisées dans tout le reste du sous-programme.

Lorsque LECT se termine sur échec, les valeurs de BC et IC sont perdues. Un nouvel appel de LECT reprendra donc le texte source à partir du même endroit.

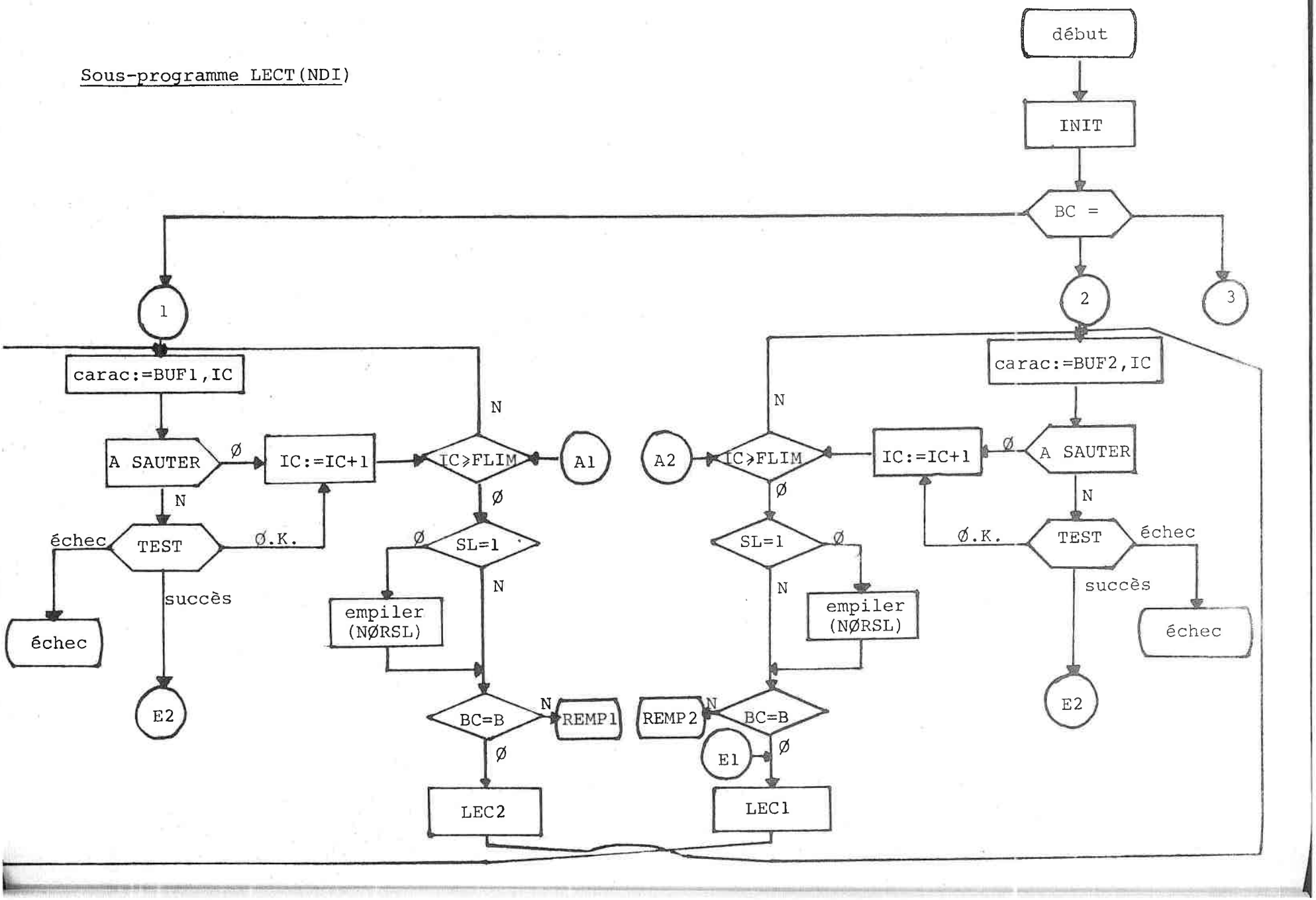
Lorsque LECT se termine sur succès, les valeurs de BC et IC sont affectées à B et I provoquant ainsi l'avancement du ruban d'entrée.

Les valeurs limites de I et IC sont :

```
DLIM   EQU   NDEB-(NDEB/4) * 4
FLIM   EQU   DLIM + LENRU
```

lorsqu'ils adressent les deux buffers BUF1 et BUF2.

Sous-programme LECT(NDI)



Commentaires et remarques

Le test IC>FLIM permet la constatation de l'épuisement d'un buffer. On peut alors empiler sur la pile PILSAU le numéro de règle du saut de ligne (NØRSL) dans le cas où cette action est demandée dans la grammaire (SL = 1).

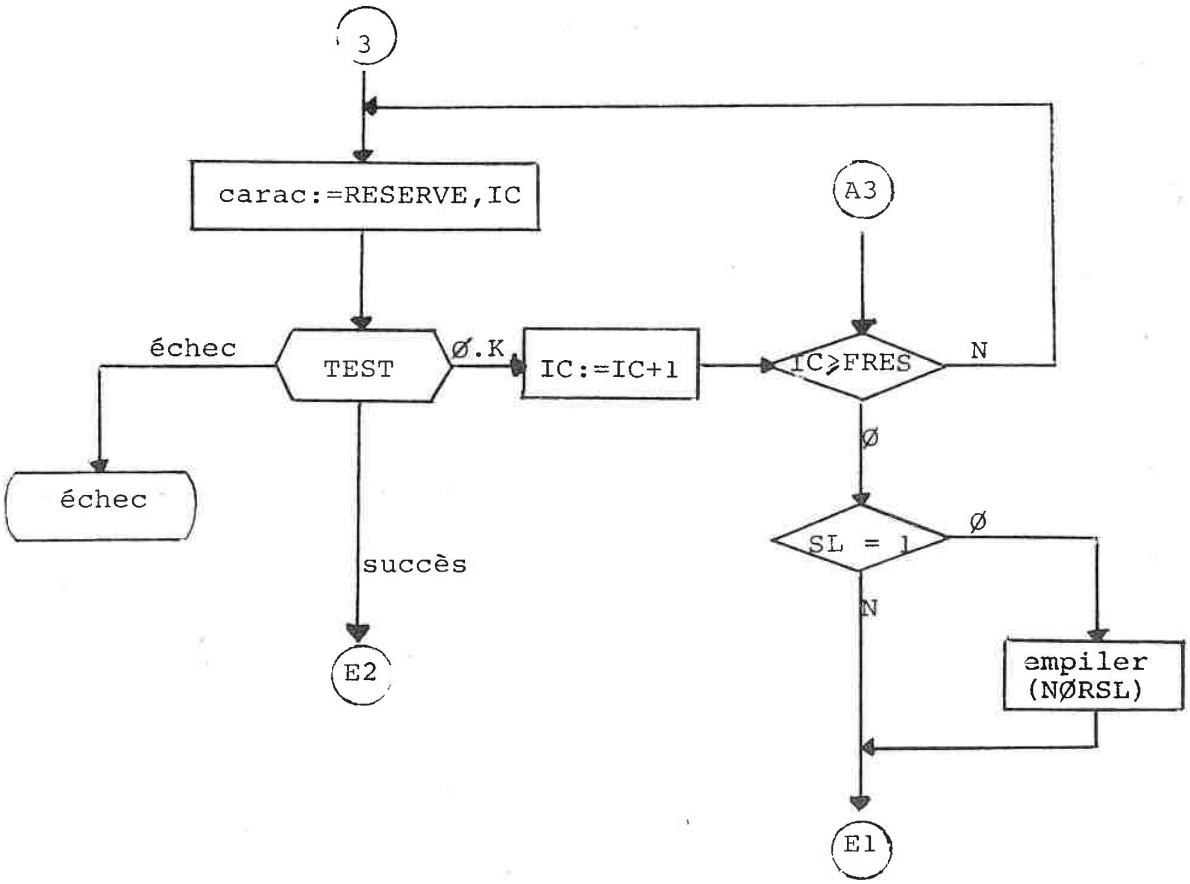
Pour poursuivre l'examen du texte source il faut utiliser un autre buffer. Dans le cas où BC = B on sait que l'autre buffer est libre et qu'il est inutile d'employer la RESERVE. On peut alors contrôler le remplissage de ces buffers au moyen de LEC1 pour B1 ou LEC2 pour B2.

Dans le cas où BC ≠ B les cas suivants peuvent se présenter : BC = 1, B = 2, les buffers B1 et B2 sont mobilisés ; la réserve n'a pas été utilisée mais doit l'être maintenant. Le contenu utile de B2 et B1 va être placé dans la RESERVE ; l'enregistrement suivant sera lu dans B1.

BC = 1, B = 3, la réserve est déjà en cours d'utilisation et le buffer B1 est plein. Le contenu de B1 va être placé dans la RESERVE et un nouvel enregistrement sera lu dans B1. Seuls la réserve et B1 sont alors utilisés, B2 ne sera réutilisé que lorsque la RESERVE sera libérée. Cela implique que la situation BC = 2, B = 3 ne peut jamais se produire.

Ces deux situations sont traitées par le module REMP1.

BC = 2, B = 1, les buffers B1 et B2 sont mobilisés ; la RESERVE n'a pas encore été utilisée mais doit l'être maintenant ; le contenu utile de B1 et B2 va être placé dans la RESERVE par le module REMP1.



Notes de programmation

Pour des raisons évidentes d'efficacité, carac sera un registre contenant le caractère à examiner dans le dernier octet, les trois premiers étant à zéro.

L'action "empiler" se fait sur la pile PILSAU qui est une pile statique. Il faut prévoir un test de débordement ⟶ message d'erreur et abandon du travail.

LE MODULE INIT

IC := I ;
BC := B ;
ADT := adresse du terminal NDI dans la TDT ;
positionnement d'un aiguillage selon le type du terminal (chaîne ou automate), et initialisation des variables rémanentes du sous-programme TEST ;

PILSAUC := PILSAU co PILSAUC est le double mot d'état de la pile où sont empilés les numéros de règles correspondant aux caractères sautés. PILSAU est le mot d'état conservé d'un appel à l'autre de LECT. PILSAU reçoit la valeur de PILSAUC en cas de succès.

LE MODULE ASAUTER (reconnait si carac est un caractère à sauter)

```
début pour RX1:=0 pas 2 jusqu'à 2 × (NSAU-1) faire
      début CH, carac *ASAU,RX1
            si égal alors début RX1:=RX1+1 ;
                                norègle:=*ASAU,RX1
                                si norègle≠-1 alors
                                        empiler(norègle)
                           allera sortie Ø du module ASAUTER
                        fin
      fin ;
      allera sortie N du module ASAUTER
fin
```

LE SOUS-PROGRAMME TEST

Il reçoit dans carac un caractère du texte source qui n'est pas un caractère à sauter. Il doit décider si oui ou non le caractère appartient au terminal NDI, trois cas peuvent se produire :

- le terminal a été reconnu et le caractère contenu dans carac n'appartient pas à ce terminal. C'est la sortie succès de TEST.

- le caractère contenu dans carac appartient au terminal dont la reconnaissance est en cours. Le caractère suivant du texte source doit être examiné. C'est la sortie Ø.K. de TEST (qui assure que LECT le rappellera sur le caractère suivant).

- le terminal n'a pas été reconnu et le caractère qui se présente n'appartient pas à ce terminal. C'est la sortie échec de TEST.

TEST peut avoir à traiter deux sortes de terminaux : les terminaux chaînes de caractères et les terminaux automates. En conséquence le sous-programme TEST rassemble en fait 2 programmes spécialisés :

TESTS pour les terminaux chaînes de caractères.

TESTLEX pour les terminaux automates.

Le choix entre TESTS et TESTLEX est fonction d'un aiguillage positionné par INIT.

a) Sous-programme TESTS

La variable ADT contient l'adresse du terminal chaîne. L'index IPT permet d'adresser chaque caractère du terminal. IPT est initialisé à zéro par INIT.

```
procédure TESTS ;
 début si  *ADT,IPT = 0 alors co le terminal a été reconnu co
                                  sortie succès ;
       si  carac ≠ *ADT,IPT alors sortie échec ;
       IPT:=IPT + 1 ;
       sortie Ø.K.
 fin
```

b) Sous-programme TESTLEX

TESTLEX est en fait chargé d'exécuter une transition de l'automate correspondant au terminal NDI.

TESTLEX utilise une variable ETAT pour mémoriser l'état de l'automate. Cette variable est initialisée au début de LECT par INIT avec le numéro de l'état initial de l'automate qui se trouve dans la TDT. INIT initialise aussi à zéro une variable LGUR qui, au moment de la sortie succès,donnera la longueur en caractères du terminal reconnu. Le terminal est placé au fur et à mesure de sa reconnaissance au sommet de la pile générale de FACE afin d'être passé en paramètre à la procédure lexicographique. Au moment de la sortie succès, la longueur LGUR est aussi placée sur la pile.

Le mot du langage régulier que doit analyser TESTLEX ne se termine pas par un délimiteur précis. En fait TESTLEX poursuit l'analyse jusqu'à la rencontre d'un caractère faisant tomber l'automate en échec (état 0). Il regarde alors si l'état précédent est un état final.

```
procédure TESTLEX ;
début ETAT1:=AUTØMATE[ETAT,CARAC];
      si ETAT1≠0 alors début ETAT:=ETAT1 ;
                             empiler carac sur la pile des paramètres
                             lexicographiques ;
                             LGUR:=LGUR+1 ;
                             sortie Ø.K. de TESTLEX
                        fin ;
      co ETAT1 = 0 ; on ne doit donc plus poursuivre l'exécution
         de l'automate. Le résultat de cette exécution dépend de
         l'état précédent mémorisé dans ETAT co
      si ETAT est un état final et LGUR≠0 alors
          début empiler LGUR sur la pile des paramètres lexicogra-
                phiques ;
                proc:=(ADT+1) co proc:=index de la procédure
                              lexicographique prélevé dans la
                              table des terminaux co
                si proc≠0 alors début appel de la procédure lexi-
                                      cographique
                                      RLEX(proc) ;
                                      RES:=résultat de la procédure
                                      lexicographique (prélevé au
                                      sommet de la pile) auquel est
                                      adjoint une marque dans le
                                      premier octet indiquant qu'il
                                      s'agit d'une feuille ;
                                      sortir (RES)
                                  si l'on est dans ANALYSE (et
                                      non dans ANAPPLYSE)
                                      alors retirer le résultat du
                                            sommet de la pile
                              fin
                         sinon la chaîne reconnue est retirée de
                                                    la pile ;
```

sortie succès de TESTLEX

fin ;

retirer la chaîne analysée du sommet de la pile ;

sortie échec de TESTLEX

fin

Remarque :

TESTLEX refuse de reconnaître la chaîne vide comme un terminal même si l'état initial de l'automate est un état final.

LE MODULE LEC1

Le module LEC1 est appelé lorsque ayant épuisé le buffer B2 ou la RESERVE on veut pouvoir utiliser le buffer B1. LEC1 doit normalement effectuer une lecture (lecture de l'enregistrement suivant) sur le DCB M:SI.

Cependant à cause des allers et retours que peuvent provoquer diverses tentatives de reconnaissance de terminaux, l'enregistrement peut déjà avoir été lu. LEC1 s'aperçoit de ce fait en testant l'indicateur B12 qui est à vrai dans le cas où la lecture a déjà été faite. L'indicateur B12 est géré à la fois par LEC1 lui-même et par la séquence de fin de traitement de LECT en cas de succès (séquence d'instructions (E2)).

LEC1 traite aussi la fin de fichier en générant un caractère spécial de fin de données dans le buffer (caractère EØD qui pourra être défini par exemple par : EØD EQU X'OO').

```
début si B12 alors allera CFAIT1 ;
      lecture d'un enregistrement sur SI dans B1, enregistrement
      de longueur LENR ; en cas de fin de fichier allera EØD1
  finlect1 : B12 := vrai  ;
  CFAIT1 : IC:=DLIM ; BC:=1 ; retour ;
  EØD1 : remise à blanc de B1
         LI,R EØD (marque de fin de données)
         STB,R BUF1,DLIM ; allera finlect1 ;
fin
```

LE MODULE LEC2

Le module LEC2 est chargé de contrôler le remplissage du buffer B2.

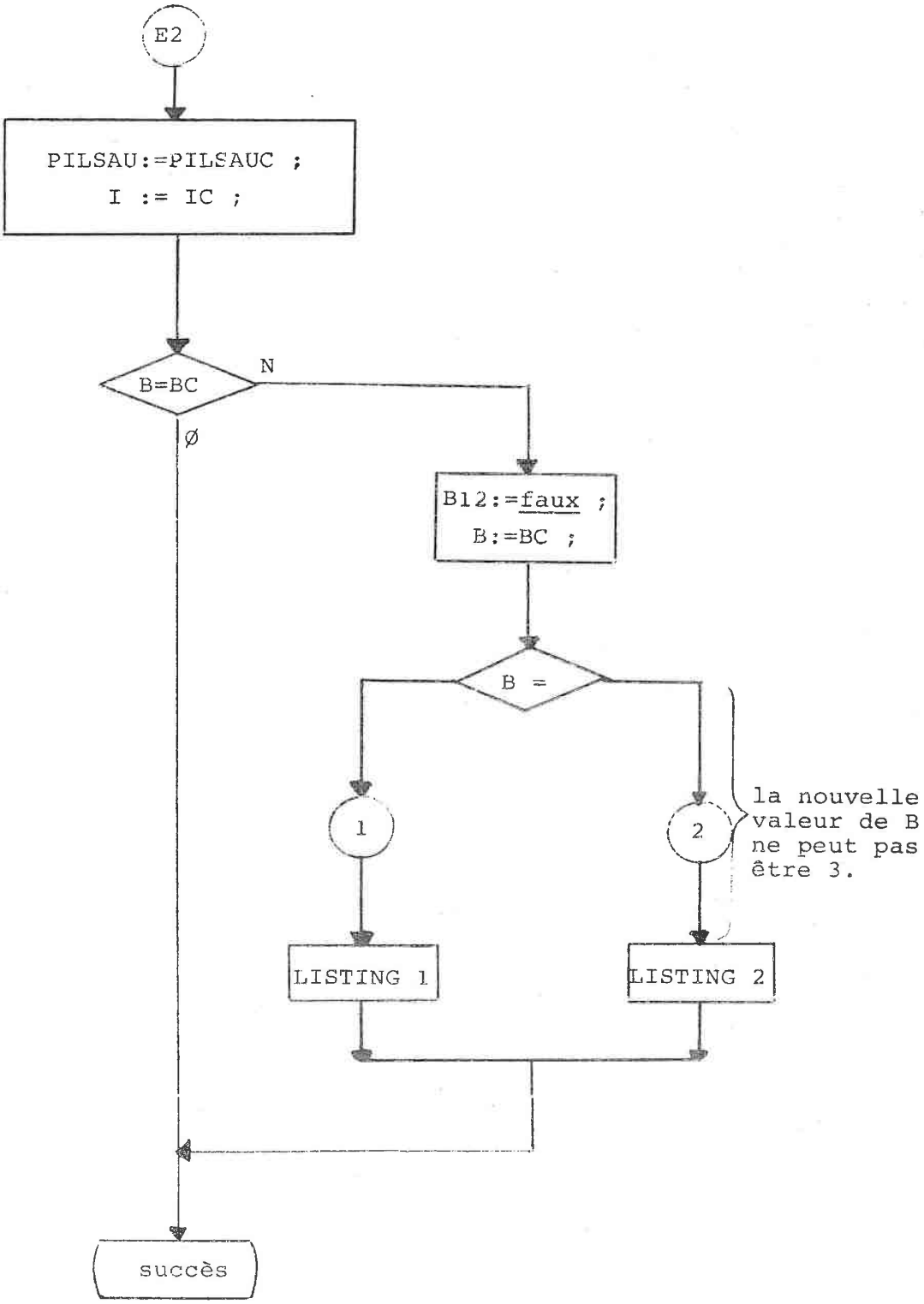
Il se déduit facilement du module LEC1 :

```
début si B12 alors allera CFAIT2 ;
          lecture d'un enregistrement sur SI dans B2,
          enregistrement de longueur LENR ; en fin de fichier
          allera EØD2 ;
     finlect2 : B12:=vrai ;
     CFAIT2   : IC:=DLIM ;
                BC:=2 ;
                retour
     EØD2 : remise à blanc de B2
            LI,R   EØD    (marque de fin de données)
            STB,R  BUF2,DLIM ; allera finlec2 ;
fin
```

L'indicateur B12 est commun à LEC1 et LEC2.

LA SORTIE SUCCES DE LECT (E2)

La reconnaissance du terminal a été faite. On doit donc affecter aux variables rémanentes de LECT (B,I,PILSAU) les valeurs des variables locales conditionnelles correspondantes. De plus si B a changé de valeur (B ≠ BC), on peut abandonner l'ancien buffer utilisé en le rendant disponible (B12 := faux) et effectuer le listing du nouvel enregistrement lu :


E2
PILSAU:=PILSAUC ;
I := IC ;
B=BC
N
Ø
B12:=faux ;
B:=BC ;
B =
1
2
la nouvelle valeur de B ne peut pas être 3.
LISTING 1
LISTING 2
succès

LE SOUS-PROGRAMME MESSERR(NO)

Emission d'un message d'erreur

Ce sous-programme est appelé par l'analyseur lorsque celui-ci détecte une erreur de syntaxe dans le texte source. C'est un sous-programme complétement indépendant de LECT, mais il intervient dans l'organisation du listing. C'est pourquoi il est étudié ici en même temps que les sous-programmes LISTING1 et LISTING2.

MESSERR reçoit en paramètre (NØ) le numéro de l'erreur (c'est un numéro de règle). Ce numéro est en binaire. Le but de MESSERR est d'imprimer au-dessous de la ligne erronée un message du genre :

**** ERREUR NØ ....

Dans le cas où l'analyse effectue un listing du texte source (LS = vrai), il suffit d'imprimer le message pour qu'il apparaisse immédiatement après la ligne erronée.

Dans le cas où l'analyse n'effectue pas de listing, il faut imprimer la ligne erronée avant d'émettre le message. Cependant s'il y a plusieurs erreurs sur une même ligne, on risque d'imprimer plusieurs fois la même ligne.

De plus on peut désirer souligner le caractère erroné dans le texte source. Là aussi dans le cas où il y a plusieurs erreurs par ligne on risque une présentation peu élégante. Il faudrait alors souligner la ligne source une fois (de plusieurs caractères) puis émettre les messages.

Pour toutes ces raisons, le sous-programme MESSERR, n'effectuera pas l'impression du message d'erreur, mais se contentera de cumuler les demandes dans une pile (PILERR). Cette pile sera exploitée par LISTING1 et LISTING2 et remise à zéro par eux.

Chaque élément de PILERR contiendra deux renseignements :
- le numéro de l'erreur
- l'indice du caractère à souligner (dans BUF1 ou BUF2) ; c'est la valeur actuelle de la variable I de LECT.

Ces renseignements peuvent fort bien tenir sur 1 demi-mot ; les éléments de PILERR peuvent donc être des mots.

Dans le cas où la RESERVE est en cours d'utilisation (B = 3) il est impossible de souligner le caractère erroné. On placera alors dans PILERR non pas la valeur de I, mais une valeur INSL (indice de non soulignement) qui provoquera le rangement du caractère de soulignement dans un octet non imprimé (le buffer utilisé par LISTING1 et LISTING2 pour préparer les impressions est ZØNECR ; sa longueur utile est de 120 octets ; ZØNECR contiendra 121 octets et INSL sera l'indice du 121ème).

```
INSL  EQU   121 - (16 + NDEB)
```

La pile PILERR est statique et doit être réservée avec une certaine dimension. Le nombre de messages d'erreurs dans une ligne peut être majoré par le nombre de caractères de cette linge. Le nombre maximum d'éléments à réserver dans PILERR est donc LENRU. De toute façon, le mot d'état PILERR sera tel qu'un éventuel débordement positionnera le code condition mais ne provoquera pas de déroutement. Cependant le code condition ne sera pas testé. Ainsi au cas où un débordement se produit les derniers messages de la ligne seront tout simplement perdus. (Ce qui n'est pas grave : quand il y a plus de 10 erreurs sur une ligne, une de plus ou une de moins !)

```
procédure MESSERR(NØ) ; entier NØ ;
    début si non LS et PILERR vide alors
            début mouvementer (si B=2 alors B2 sinon B1) sur LENR
                  octets dans ZØNECR + 16 octets ;
                  éditer NØLGN sur 8 caractères avec suppression
                  des zéros non significatifs dans ZØNECR + 4 ;
                  imprimer(ZØNECR)
            fin ;
```

```
            co Dans le cas où la RESERVE est en cours d'utili-
               sation (B = 3) la ligne imprimée en l'absence
               de listing n'est qu'indicative de l'endroit
               approximatif de l'erreur
            co
        1er demi-mot de R1:=NØ ;
        si B = 3 alors 2ème demi-mot de R1:=INSL
                 sinon 2ème demi-mot de R1:=I ;
        empiler R1 sur PILERR ;
        CTERR:=CTERR + 1  co comptage du nombre d'erreurs co
    fin
```

LES MODULES LISTING1 et LISTING2

Les modules LISTING1 et LISTING2 sont chargés d'effectuer le listing du contenu des buffers B1 et B2 en assortissant cette ligne d'un numéro de ligne (NØLGN) qu'ils gèrent.

En fait le listage dépend d'une variable LS (LS = vrai un listing est demandé, LS = faux le listing n'est pas demandé). La variable LS est positionnée dans le PREAMBULE de l'analyse selon le résultat de l'analyse de la carte de commande.

De plus ces modules devront examiner la pile PILERR et effectuer la présentation des messages d'erreur qu'elle contient.

On peut envisager aussi une gestion du saut de page, mais le plus simple est d'inclure cela dans le DCB de sortie.

Toutes les lignes imprimées sont d'abord préparées dans ZØNECR (zone de 31 mots). Les lignes imprimées ont une longueur de 120 octets.

En fait les modules LISTING1 et LISTING2 sont identiques à ceci près que LISTING1 liste le contenu de B1 et LISTING2 liste le contenu de B2. Il est inutile de les programmer 2 fois. On peut très bien avoir un sous-programme à paramètre. Ici on listera le contenu de BUF.

```
          BUF = si LISTING1 alors B1 sinon B2.

début NØLGN:=NØLGN + 1 ;
      si PILERR non vide alors
            début mettre ZØNECR à blanc ;
               pour chaque élément R1 de  PILERR faire
               début indice:=2ème demi-mot de R1
                     ZØNECR[16+NDEB+indice]:=C'I'
                              co soulignement des erreurs co
               fin ;
               imprimer(ZØNECR) ;
               pour chaque élément R1 de PILERR faire
               début no:=1er demi-mot de R1 ;
                     éditer(si no=-1 alors blancs sinon no en
                     décimal avec suppression des zéros non
                     significatifs dans MESSAGE) ;
                     imprimer(MESSAGE)
               fin ;
               vider la pile PILERR ;
            fin ;
      si LS alors début mouvementer BUF sur une longueur de LENR
                        octets dans ZØNECR + 16 octets ;
                        éditer NØLGN dans ZØNECR + 4 octets sur
                        8 octets avec suppression des zéros non
                        significatifs ;
                        imprimer ZØNECR
                  fin
fin

MESSAGE     DATA     C' **** ERREUR NØ '
            RES      5C       pour éditer le numéro de l'erreur.
```

NØLGN est initialisé à zéro dans PREAMBULE.

LES MODULES REMP1 et REMP2

Ces modules sont appelés lorsqu'il est nécessaire d'utiliser la RESERVE.

Lorsque REMP2 est appelé on est dans la situation B = 1, BC = 2, la RESERVE n'a pas été utilisée. On range la partie utile de BUF1 (depuis l'indice I) et la totalité de BUF2 dans la RESERVE (cela se fait à l'aide du sous-programme REMPLIR).

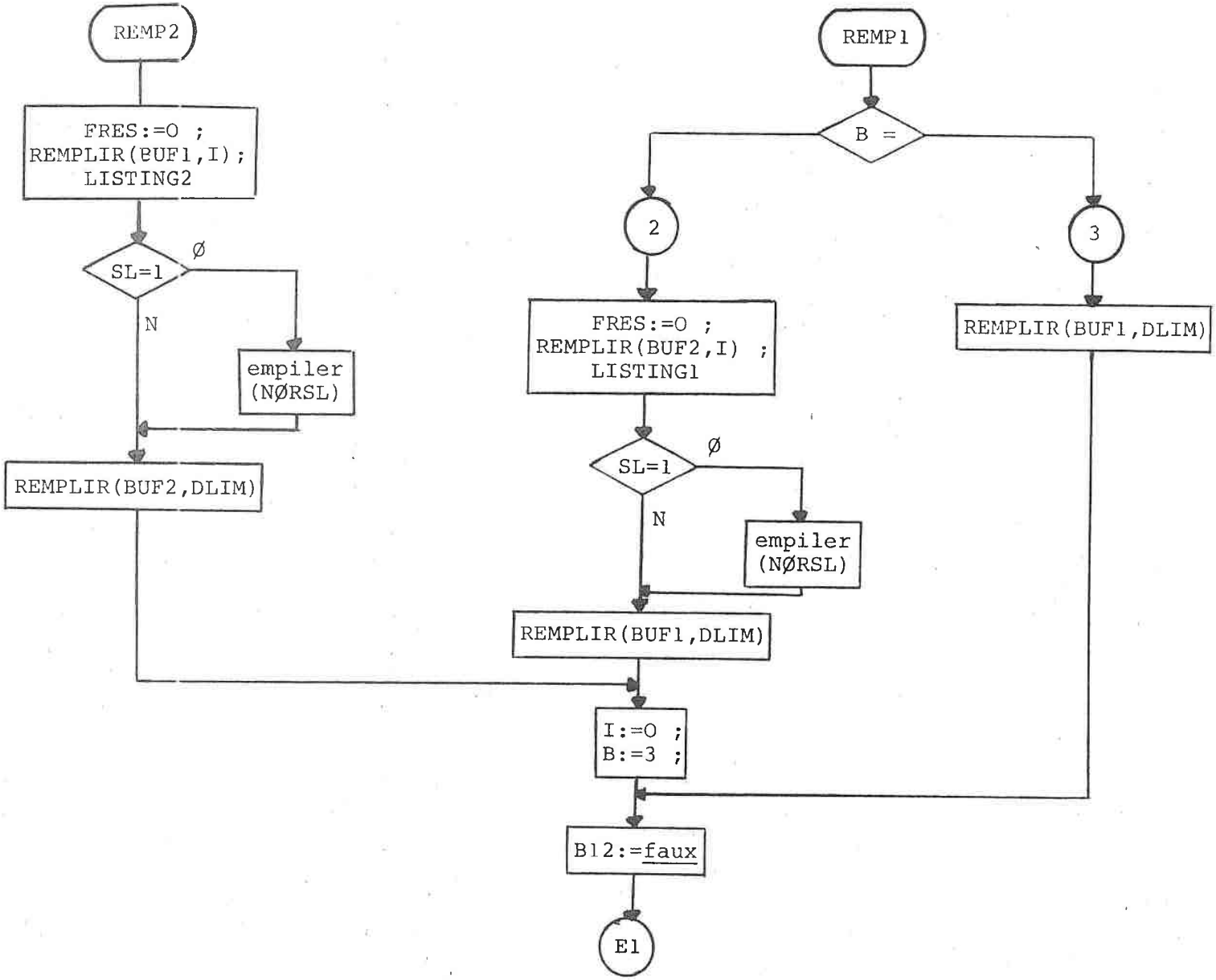
Lorsque REMP1 est appelé, BC = 1 et B peut être égal à 2 ou 3. Dans le cas où B = 2, la situation est symétrique à celle de REMP2. Dans le cas où B = 3, il faut simplement vider BUF1 dans la RESERVE et faire une nouvelle lecture dans B1.

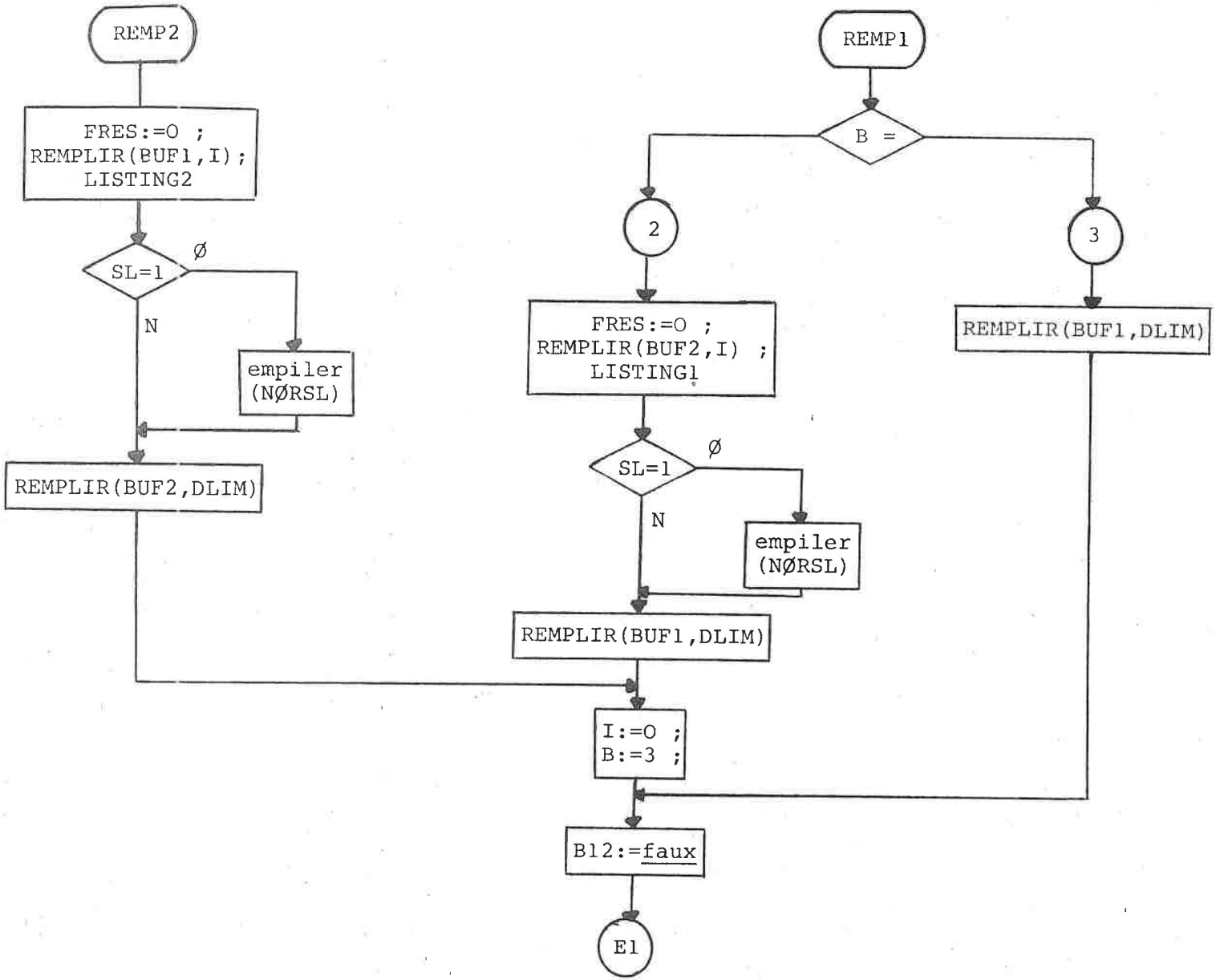
Le sous-programme REMPLIR est utilisé pour remplir la RESERVE. Il a deux paramètres :

- l'adresse du buffer à utiliser.
- l'indice à partir duquel il faut vider ce buffer.

Note de réalisation

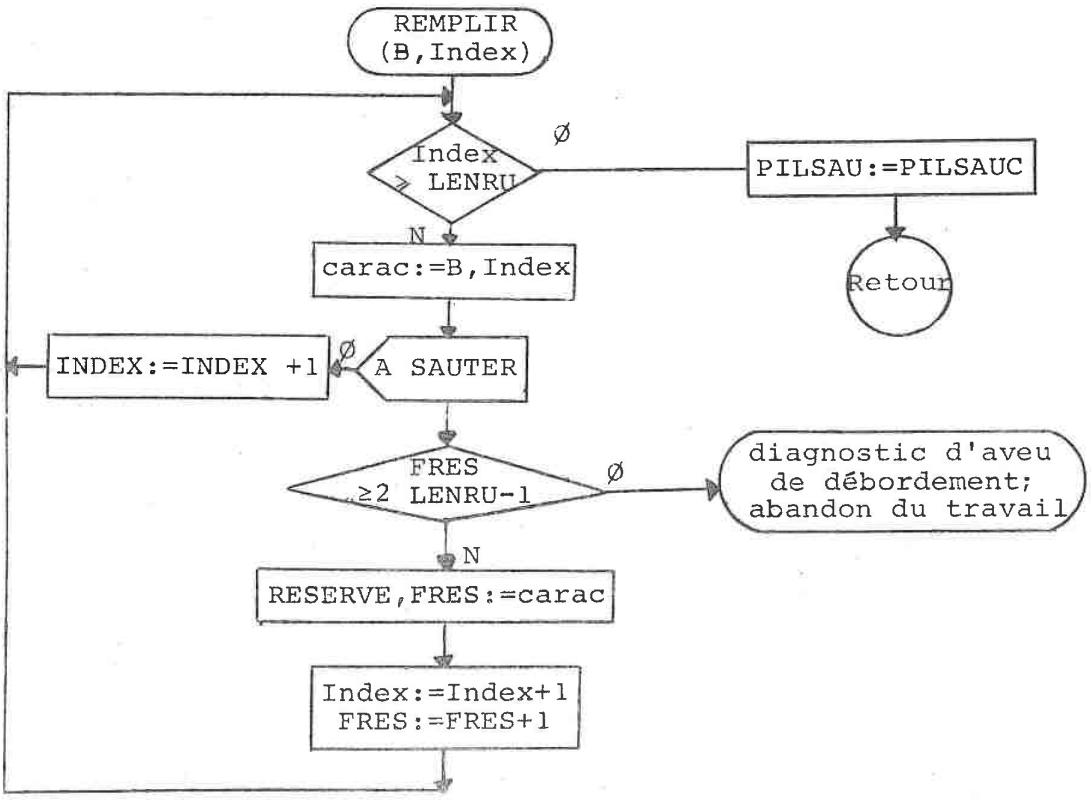
L'utilisation de la RESERVE au moyen des modules REMP1 et REMP2 est une procédure de secours (qui amène d'ailleurs un inconvénient comme il est expliqué dans l'étude du sous-programme REMPLIR) utile lorsque l'on trouve dans le texte source une densité particulièrement importante de caractères à sauter. Dans une première phase de réalisation du sous-programme LECT, il est fort possible de ne pas inclure cette procédure de secours et de remplacer les modules REMP1 et REMP2 par une séquence d'instructions émettant un message et provoquant l'abandon du travail.


REMP2
FRES:=O ;
REMPLIR(BUF1,I);
LISTING2
SL=1
Ø
N
empiler
(NØRSL)
REMPLIR(BUF2,DLIM)
REMP1
B =
2
3
FRES:=O ;
REMPLIR(BUF2,I) ;
LISTING1
SL=1
Ø
N
empiler
(NØRSL)
REMPLIR(BUF1,DLIM)
REMPLIR(BUF1,DLIM)
I:=O ;
B:=3 ;
B12:=faux
E1


REMP2
FRES:=O ;
REMPLIR(BUF1,I);
LISTING2
SL=1
Ø
N
empiler
(NØRSL)
REMPLIR(BUF2,DLIM)
REMP1
B =
2
3
FRES:=O ;
REMPLIR(BUF2,I) ;
LISTING1
SL=1
Ø
N
empiler
(NØRSL)
REMPLIR(BUF1,DLIM)
REMPLIR(BUF1,DLIM)
I:=O ;
B:=3 ;
B12:=faux
E1

LE SOUS-PROGRAMME REMPLIR(B,Index)

Le rôle du sous-programme REMPLIR est de placer le contenu de BUF1 ou BUF2 dans la RESERVE en faisant disparaître les caractères à sauter. Lorsque le contenu de la RESERVE sera examiné par d'éventuels appels ultérieurs de LECT, les numéros de règles correspondant à ces caractères ne pourront plus être générés. Pour remédier à cet inconvénient, ces règles seront placées dans la pile PILSAU par REMPLIR de manière définitive, c'est-à-dire en affectant au mot d'état PILSAU la valeur de PILSAUC. Il peut en résulter que ces règles ne soient pas générées exactement à l'endroit souhaité de la chaîne postfixée. Mais il faut considérer que les procédures REMP1 et REMP2 sont des procédures de secours utilisées lorsque le texte à analyser contient une densité anormalement élevée de caractères à sauter. Il est normal que cette procédure exceptionnelle entraîne quelque inconvénient.



LES INITIALISATIONS NECESSAIRES A L'UTILISATION DES SOUS-PROGRAMMES LECT et MESSERR

Les sous-programmes LECT et MESSERR étant communs aux deux analyseurs, les initialisations nécessaires à leur utilisation peuvent être faites dans PREAMBULE.

PREAMBULE positionne la variable LS selon la carte commande qu'il reçoit.

De plus le premier enregistrement du texte source doit être lu et certaines variables positionnées :

```
B12:=faux ;
LEC1;
B12:=faux ;
NØLGN:=0 ; PILERR:=vide;
LISTING1 ; co de manière à faire le listing éventuel
           du premier enregistrement co
B:=1 ; I:=DLIM ;
CTERR:=0 ;
```

A la fin du traitement il se peut que le dernier enregistrement lu ne soit pas listé. Il faut alors qu'il le soit. Dans le cas où le listing de la dernière carte a été fait, on listera alors l'enregistrement de fin de fichier contenant des blancs, créé par LEC1 ou LEC2 ; c'est donc sans inconvénient. Il suffit de faire :

```
B:=BC ; si B = 1 alors LISTING1 sinon LISTING2
```

LE SOUS-PROGRAMME AVANCCAR

Ce sous-programme permet d'avancer de 1 caractère dans le texte source. Il est utilisé par les programmes de reprise de l'analyse en cas d'erreur.

Le sous-programme AVANCCAR doit modifier la variable I de LECT (incrémentation de 1). Cette incrémentation de 1 peut éventuellement provoquer une nouvelle lecture physique et une commutation de buffer. De plus il est nécessaire de continuer à sauter les caractères à sauter. La solution la plus simple pour effectuer cette gestion consiste à utiliser celle qui est faite dans le sous-programme LECT lui-même. C'est pourquoi, le sous-programme AVANCCAR constitue en fait un nouveau point d'ENTREE du sous-programme LECT plutôt qu'un sous-programme indépendant. Des variables de LECT sont simplement positionnées pour qu'aucun terminal ne soit jamais reconnu.

Il s'écrit :

```
début ADT:=adresse d'un mot contenant zéro
          co adresse d'un terminal ne contenant aucun caractère co;
      IPT:=0 co index de parcours d'un terminal du sous-programme
                TEST de LECT co ;
      positionnement de l'aiguillage de LECT sur la lecture d'un
      terminal chaîne (module TESTS et non TESTLEX) ;
      I:=I+1 ; BC:=B ; IC:=I ;
      PILSAUC:=PILSAU ;
      si BC=1 alors allera A1 ;
      si BC=2 alors allera A2 ;
      allera A3
fin
```

A N N E X E 3

Etude du sous-programme REØRG

(réorganisation de la mémoire après construction
de la représentation ascendante de la grammaire)

```
entier tableau DESTAB,NDESTAB[1:14] ;
début AVECT:=ATDT + LTDT + 1 co AVECT=adresse de la liste de
                                vecteurs de renseignements co;
      ATGA:=AVECT + 18        co ATGA:=adresse de début du TGA ;
                                22 est la taille de la liste
                                de vecteurs de renseignements
                                co ;
      ranger en *ATDT,LTDT un terminal supplémentaire : la marque
                           de fin de fichier EØD de LECT ;
      DESTAB[1]:=INT ;
      DESTAB[2]:={INT + 1} ;
      DESTAB[3]:=IN ; DESTAB[4]:={IN + 1} ;
      DESTAB[5]:=FIN ; DESTAB[6]:={FIN + 1} ;
      DESTAB[7]:=PRECED1 ; DESTAB[8]:={PRECED1 + 1} ;
      DESTAB[9]:=PRECED2 ; DESTAB[10]:={PRECED2 + 1} ;
      DESTAB[11]:=AENTREE ; DESTAB[12]:=LTDT + LTMS + 1 ;
      DESTAB[13]:=VDR ; DESTAB[14]:={VDR + 1} ;
      MEMLIB:=si INT ≤ {PILDEPIL + 2} alors INT - ATGA
                                      sinon {PILDEPIL + 2}  - ATG1 + 1
                                      + 1er demi-mot de {PILDEPIL + 3}
      co MEMLIB représente la taille en mots de l'espace mémoire
      libre compris entre ATGA, c'est-à-dire après la liste de
      vecteurs de renseignements et le tableau INT ; dans la
      mesure où INT ne se trouve pas dans la 1ère section de la
      pile, il faut tenir compte de la place encore disponible
      dans cette section de pile (compte des espaces du mot d'état
      de la section de pile)co ;

      si MEMLIB < LTGA alors KSPEC
                       sinon KNØRMAL

fin
```

ETUDE DU MODULE KNØRMAL

On sait qu'il y a suffisamment de place dans la 1ère section de pile et avant le début du tableau INT pour placer le TGA. On va donc mémoriser le TGA à la place du GRG puis les six

tables décrites par DESTAB seront décalées vers la basse mémoire immédiatement derrière le TGA.

```
KNØRMAL :
début AD:=ATGA co AD = adresse 1ère mémoire libre après le TGA
                       ou la partie du TGA mise en mémoire co ;
      si le fichier de travail est fermé alors allera dernenr ;
      fermer le fichier de travail ;
      ouvrir le fichier de travail en entrée ;
  lec1 : lire co 1 enregistrement doit être placé en *AD ; la
                 longueur en mots dans R1 ; en fin de fichier
                 allera fermer co ;
      AD:=AD + R1 ;
      allera lec1 ;
  fermer : fermer GRAM ;
  dernenr : mouvementer le contenu de ZBUF[0 : IB - 1] en *AD
            co attention : IB peut être égal à zéro co ;
            RX1:=T co adresse de début de la grammaire co ;
            3[RX1]:=ATGA-RX1 co adresse relative du TGA co ;
            4[RX1]:=LTGA ; AD:=AD+LBUF ;
            MEMLIB:={PILDEPIL+2} - AD + 1 + 1er demi-mot de
                                          {PILDEPIL+3} ;
            AR:=ATGA+LTGA-RX1 co AR = adresse relative de la
                                      zone mémoire non affectée
                                      co ;
  nsec:=2 co adresse relative du mot d'état de la section de
            pile par rapport à PILDEPIL ; nsec=2 désigne la
            première section de pile, les 2 premiers mots
            étant le descripteur de PILDEPIL co

pour i:=1 pas 2 jusqua 13 faire
début co traitement effectué pour chacune des 7 tables construi-
        tes à la suite de TGA co.
      NDESTAB[i]:=AR ; AR:=AR+DESTAB[i+1] ;
      co le tableau numéro i doit être déplacé en mémoire. Il
         faut envisager plusieurs cas co ;
```

```
deb:si (PILDEPIL[nsec] + 1er demi-mot de PILDEPIL[nsec + 1])
                                                    < DESTAB[i]
    alors co le tableau numéro i ne se trouve pas dans la
            section de pile courante, celle que l'on est en
            train de remplir co
        début si MEMLIB ≥ DESTAB[i + 1]alors co il y a assez de
                    place dans la section de pile courante pour
                    y placer le tableau numéro i co
                    début mouvementer  *DESTAB[i] en *AD sur
                            DESTAB[i + 1] mots ;
                            DESTAB[i] :=AD co nouvelle adresse
                                        du tableau co ;
                            AD:=AD + DESTAB[i + 1] ;
                            MEMLIB:=MEMLIB - DESTAB[i + 1] ;
                            allera finpour
                    fin ;
                co il n'y a pas assez de place dans la section
                   courante pour y placer le tableau numéro i,
                   il faut alors mettre à jour le mot d'état de
                   la section de pile et passer à la section
                   suivante co
                   R:=-MEMLIB + 1er demi-mot de PILDEPIL[nsec + 1] ;
                   MSP,R PILDEPIL,nsec  co mise à jour du mot
                                           d'état de la section de
                                           pile co
                   nsec:=nsec + 2  co passage à la section de pile
                                       suivante co ;
                   AD:=1 + PILDEPIL[nsec] - 2ème demi-mot de
                                       PILDEPIL[nsec + 1]
                co AD = adresse de début de la nouvelle section
                       de pile co ;
                MEMLIB:=1er demi-mot de PILDEPIL[nsec + 1]
                       + 2ème demi-mot de PILDEPIL[nsec + 1] ;
                allera deb
        fin ;
```

```
        co Le tableau numéro i se trouve dans la section de
           pile courante ; on est sûr dans ce cas que la place
           disponible est suffisante. Il faut simplement le
           translater vers la basse mémoire co
        si AD=DESTAB[i] alors co translation inutile co allera
                                svt ;
        mouvementer *DESTAB[i] en *AD sur une longueur de
        DESTAB[i + 1] mots ;
  svt : DESTAB[i]:=AD ;
        AD:=AD + DESTAB[i + 1] ;
        MEMLIB:=MEMLIB - DESTAB[i + 1] ;
finpour : fin ;
  co les tableaux sont translatés en mémoire ; il reste à mettre
     à jour les vecteurs de renseignements co
       RX1:=T co adresse de début de la grammaire co
INT:=7[RX1]:=DESTAB[1] ; 8[RX1]:={INT + 1} ;
     9[RX1]:={INT + 2} ; 10[RX1]:={INT + 3} ;
     1[RX1]:= DESTAB[11] ;
     RX1:=AVECT ; VDR:=DESTAB[13]
     IN:=0[RX1]:=DESTAB[3] ; 1[RX1]:={IN + 1} ;
     2[RX1]:={IN + 2} ; 3[RX1]:={IN + 3} ;
     FIN:=4[RX1]:=DESTAB[5] ; 5[RX1]:={FIN + 1} ;
     6[RX1]:={FIN + 2}; 7[RX1]:={FIN + 3} ;
     PRECED1:=8[RX1]:=DESTAB[7] ; 9[RX1]:={PRECED1 + 1} ;
     10[RX1]:={PRECED1 + 2} ; 11[RX1]:={PRECED1 + 3} ;
     PRECED2:=12[RX1]:= DESTAB[9] ; 13[RX1]:={PRECED2 + 1} ;
     14[RX1]:={PRECED2 + 2} ; 15[RX2]:={PRECED2 + 3} ;
     co les tables sont rangées au bon endroit. Les vecteurs de
        renseignements sont construits ; il ne reste plus qu'à
        mettre à jour la pile MEM pour libérer de la mémoire co;
     pour i:=2ème demi-mot de {PILDEPIL + 1} et X'7FFF' pas -2
     jusqua nsec + 2 faire
     début co séquence d'instructions exécutée pour chaque section
              de pile totalement libérée (nsec est le numéro de la
              dernière section actuellement occupée) co
              R:=-(2ème demi-mot de PILDEPIL[i + 1]) ;
              MSP,R MEM co libération de l'espace occupé dans
                          cette section co
     fin ;
```

```
    co Dans la dernière section de pile occupée il faut libérer
       la partie non occupée co
    R:=AD - PILDEPIL[nsec] -1 ;
    MSP,R MEM ;
    PLW,R MEM ;  }
    PSW,R MEM ;  } pour libérer éventuellement une section.
    RETURN
fin
```

ETUDE DU MODULE KSPEC

Le module KSPEC est utilisé pour réorganiser la mémoire lorsque l'espace libre dans la 1ère section de pile est insuffisant pour recevoir le TGA (ce qui en principe doit être rare).
Deux sous-cas peuvent se présenter :

- ou bien l'espace mémoire compris entre ATGA et la fin de la 1ère section est insuffisant. Ce cas ne sera pas traité. Il conduira à un message d'erreur. Au cas où il se produit, on peut le corriger en allouant en FACE un espace plus grand à la zone des grammaires.

- ou bien l'espace libre est insuffisant à cause de la présence dans la 1ère section d'une ou plusieurs tables construites après le GRG. Il est alors nécessaire de translater ces tables vers la haute mémoire.

Le premier rôle du module KSPEC est donc d'évaluer dans MEMLIB la taille de mémoire disponible dans la 1ère section de pile à partir du début du TGA (ATGA).

```
début MEMLIB:={PILDEPIL + 2} - ATGA + 1 + 1er demi-mot de
              {PILDEPIL + 3} ;
      si MEMLIB < LTGA alors co il est impossible de placer le
                               TGA dans la 1ère section de pile.On
                                n'envisage pas de traiter ce cas
                               actuellement dans FADA.
```

```
                                    S'il survient il faut agrandir la
                                    taille allouée aux grammaires en
                                    FACE co
                              début
                                    émettre un message "débordement
                                    du graphe" ; abandon du travail
                              fin ;
co le TGA peut être placé dans la 1ère section de pile à condition
de translater les tables construites après le GRG vers la haute
mémoire. Cette translation va se faire en 2 temps. Il faut d'abord
construire une image mémoire des tables dans NDESTAB en procédant
de la basse mémoire vers la haute mémoire, puis effectuer la trans-
lation en procédant de la haute mémoire vers la basse mémoire co
MEMLIB:=MEMLIB-LTGA co espace disponible pour les tables dans la
                       1ère section de pile co ;
AD:=ATGA+LTGA        co adresse 1ère mémoire libre aprèsle TGA co ;
nsec:=2 co numéro de section de pile en cours ; ici la première co ;
nsec1:=1er demi-mot de {PILDEPIL + 1} et X'7FFF'
co nsec1 est le numéro de section de pile limite (le plus grand)
   actuellement alloué co ;
pour i:=1 pas 2 jusqua 13 faire co boucle de construction du
                                  nouvel emplacement mémoire de
                                  chaque tableau co
début imagmem :
      si MEMLIB ≥ DESTAB[i+1]
         alors co il y a assez de place dans la section de pile
                  courante nsec pour y placer le tableau numéro i
               co
      imageval:début NDESTAB[i]:=AD co nouvelle adresse du tableau
                                       numéro i co ;
                     MEMLIB:=MEMLIB-DESTAB[i+1] ;
                     AD:=AD+DESTAB[i+1] ;
               fin
         sinon co il n'y a pas assez de place dans la section de
                  pile courante nsec pour y placer le tableau
                  numéro i co
```

```
début si nsec < nsecl
   alors co la mémoire allouée n'est pas épuisée co
         début nsec:=nsec+2 co passage à la section
                de pile suivante co ;
                MEMLIB:=1er demi-mot de PILDEPIL[nsec+1]
                + 2ème demi-mot de PILDEPIL[nsec+1]
                co toute la section est considérée
                libre co ;
                AD:=PILDEPIL[nsec] - 2ème demi-mot de
                PILDEPIL[nsec+1] + 1
                co adresse de début de la section co ;
             allera imagmem
         fin
   sinon co la mémoire actuellement allouée est
            épuisée ; il faut faire une nouvelle
            demande co
         début R:=DESTAB[i+1] co R:=nombre de mots
                nécessaires co ;
                MSP,R MEM ;
                nsecl:=1er demi-mot de {PILDEPIL + 1}
                et X'7FFF' co mise à jour de nsecl
                car une nouvelle section de pile a pu
                être allouée co ;
             si nsecl = nsec alors co on reste dans
                la même section co
                MEMLIB:=MEMLIB+R+1er demi-mot de
                PILDEPIL[nsec+1]
              sinon début MEMLIB:=R+1er demi-mot de
                PILDEPIL[nsecl+1] ;
                AD:=PILDEPIL[nsecl] - R + 1  ;
                nsec:=nsecl
         fin ;
   co AD, MEMLIB et nsec sont mis à jour compte tenu
         de la nouvelle allocation mémoire ; il est
         maintenant possible de chercher la nouvelle
         image mémoire du tableau numéro i en sachant
         qu'il est certain qu'il y a assez de place
         pour lui dans la section numéro nsec co
```

```
                allera imageval
            fin
        fin
    fin ;
    co L'évaluation des nouvelles images mémoire dans NDESTAB est
       terminée. Il faut maintenant translater les tableaux en ces
       nouveaux emplacements. Il y a une précaution à prendre : ne
       pas écraser le dernier enregistrement du TGA qui se trouve
       encore dans ZBUF. Si cela doit arriver on place le contenu
       de ZBUF sur disque co
    si AD > ZBUF alors début écrire sur disque *ZBUF[0:LBUF] ;
                                    LBUF:=0
                    fin ;
    pour i:=13 pas -2 jusqua 1 faire
    début co la mouvementation du tableau i vers la haute mémoire
            ne doit pas se faire par des instructions MBS car les
            zones d'émission et de réception peuvent se chevaucher
            co
            ADE:=DESTAB[i] ;
            ADR:=NDESTAB[i] ;
            pour j:=DESTAB[i+1] - 1 pas -1 jusqua 0 faire
            *ADR[j]:=*ADE[j] ;
            DESTAB[i]:=ADR co les nouvelles adresses sont placées
                           dans DESTAB co
    fin ;
    co il y a maintenant assez de place dans la 1ère section de
       pile pour recevoir le TGA. On se retrouve ainsi dans le
       cas normal et il suffit pour placer le TGA en mémoire de
       se brancher à KNØRMAL. La séquence de programme ainsi
       déroulée placera le TGA en mémoire ;
            construira les vecteurs de renseignements et les
            mettra à jour ;
            mettra à jour les mots d'état des diverses sections
            de pile ;
            les tables ne seront pas déplacées en mémoire
            puisque les emplacements qui seront choisis seront
            les mêmes que ceux qui viennent de l'être et cette
            identité est testée dans KNØRMAL co
    allera KNØRMAL
fin
```

A N N E X E 4

Analyse détaillée du sous-programme
CØNSEC (construction des compléments
aux matrices de précédence)

LE SOUS-PROGRAMME CØNSEC(Rx) appelé par VERIFA (cf. 6.6)

Construction d'une entrée secondaire pour le symbole non terminal RX. A l'initialisation de VERIFA la table ENTSEC est réservée et initialisée à zéro.

Pour chaque symbole RX, ENTSEC construit la liste des symboles B tels que B ≺ RX. Cette construction se fait en mémoire dynamique. Il est nécessaire que la table ENTSEC et les listes de symboles B soient placées en séquence en mémoire. CØNSEC doit donc gérer la mémoire dynamique : dans le cas où une demande de mémoire provoque l'allocation d'une nouvelle section non en séquence en mémoire, il faut translater le tableau construit dans la nouvelle section.

AR initialisé au début de VERIFA représente toujours l'adresse relative à ENTSEC du 1er mot libre. A la fin du contrôle AR représente la longueur totale des entrées secondaires. Si AR=LTMS cette longueur est nulle.

```
début MEM1:=MEM ; {MEM1+1}:={MEM+1} ou X'80008000' ;
      co MEM1 double mot d'état identique à MEM mais ne provoquant
         pas de déroutement sur débordement. A l'aide de MEM1, CØNSEC
         peut gérer l'allocation de mémoire co
      *ENTSEC,RX:=AR co mise en place de l'entrée RX dans ENTSEC co
       AMC:=AR co AMC=adresse relative du mot compteur co ;
       R:=0 ;
       PSW,R MEM1 co initialisation du mot compteur co ;
       BNSØ  ⊄+2  co branchement s'il n'y a pas de débordement supé-
                     rieur co
       BAL,2 DEBØRD1 co traitement d'un débordement de pile co ;
       MC:=0 co mot compteur du nombre de symboles B tels que B ≺ RX co
       A:=-RX-1 ;
       pour B:=-LTMS pas 1 jusqua LTDT-1 faire
       début pour j:=-LTMS pas 1 jusqua -1 faire
             début si VDR[B,j]=1 alors début si IN[j,A] alors
                   allera empilB fin
             fin ;
```

```
empilB : PSW,B MEM1 ;
         BNSØ  $+2 ;
         BAL,2 DEBØRD1 ;
         AR:=AR+1 ; MC:=MC+1
fin ;
co il faut prévoir une entrée supplémentaire pour le butoir de
   début (n° LTDT) qui sera utilisée ou non selon l'axiome choisi
   au moment de l'analyse co
R:=LTDT ; PSW,R MEM1 ;
BNSØ $+2
BAL,2 DEBØRD1 ;
RX:=ENTSEC+AMC ; O,RX:=MC co mémorisation du mot compteur co
MEM:=MEM1 ;{MEM+1}:={MEM1+1} et X'7FFF7FFF' ;
fin
```

LE SOUS-PROGRAMME DEBØRD1

Il est appelé par CØNSEC lorsque la pile MEM1 déborde. Cela signifie qu'il faut allouer de la mémoire supplémentaire. La mémoire allouée peut soit augmenter la taille de la section en cours, soit constituer une nouvelle section de pile. Dans ce dernier cas il faut,translater la partie actuellement construite d'entrées secondaires dans la nouvelle section, mettre à jour le mot d'état de l'axiome section ainsi que ENTSEC.

```
début MEM:=MEM1 ; {MEM+1}:={MEM1+1} et X'7FFF7FFF' ;
      R:=AR+2 ;
      MSP,R MEM co doit provoquer l'allocation d'une nouvelle tranche
                  de mémoire dynamique co ;
      si MEM:=MEM1+R alors co allocation faite dans la même section de
                              pile co
                          début R:=-R ;
                                MSP,R MEM co la pile MEM est remise
                                dans l'état précédant l'appel, le
                                compte des espaces seul est modifié
                                co {MEM1+1}:={MEM+1} ou X'80008000' ;
                                R2:=R2-2 co le registre 2 est le
                                registre de liaison ; il faut revenir
```

```
                                à l'instruction PSW du programme
                                appelant co ;
                                retour au programme appelant,
                                avec restauration des registres,
                                par le registre de liaison R2
                        fin ;
    co une nouvelle section est allouée ; la partie des matri-
       ces complémentaires déjà construite doit être translatée
       dans cette nouvelle section co
    ADEBS:=MEM-R+1 co adresse de début de la nouvelle section co;
    mouvementer *ENTSEC en *ADEBS sur AR mots ;
    R:=-2 ; MSP,R MEM co la pile MEM est remise dans un état
                         correspondant à celui de l'appel co ;
    ENTSEC:=ADEBS co modification de l'adresse de début co ;
    RX:=2ème demi-mot de {PILDEPIL+1} et X'7FFF' ;
    R:=-AR ; MSP,R PILDEPIL-2, RX co mise à jour du mot d'état
                                     de l'ancienne section de
                                     pile co ;
    R2:=R2-2 co le registre 2 est le registre de liaison ; il
                faut revenir à l'instruction PSW du programme
                appelant co ;
    retour au programme appelant, avec restauration des regis-
    tres, par le registre de liaison R2
fin
```

A N N E X E 5

Analyse détaillée de la 2ème partie du sous-programme VERIFA (cf. 6.6, p. VI.30).

Cette partie du sous-programme réutilise l'espace occupé par la matrice VDR devenue inutile pour y placer les compléments aux matrices de précédence

```
début
co la taille des compléments aux matrices de précédence est AR co
si AR=LTMS+LTDT alors co il n'y a pas d'entrée secondaire à prévoir
                début R:=(AR+{VDR+1}) ;
                      MSP,R MEM ;  ┐
                      PLW,R MEM ;  ├ restitution d'espace libre
                      PSW,R MEM ;  ┘
                      R:=AVECT ;
                      16[R]:=0 ; 17[R]:=0
                      allera fin ;
                fin ;
co la table des entrées secondaires doit être déplacée en mémoire
   à la suite du tableau ENTREE co
   AL:=VDR co adresse mémoire libre co ;
   i:=2 ;
cherchsec: si PILDEPIL,i < VDR alors début i:=i+2 ; allera cherchsec
                                                              fin ;
   co i est le numéro de la section contenant le tableau VDR co
   MEMLIB:=PILDEPIL,i + 1er demi-mot de {PILDEPIL+1},i + 1 - VDR
   co MEMLIB = espace libre dans cette section co ;
rang : si MEMLIB ⩾ AR alors co il y a assez de place dans la section
                              numéro i pour placer les compléments
                              aux matrices de précédence co
                       début mouvementer  *ENTSEC en *AL sur AR mots ;
                             ENTSEC:=AL ;
                             MEMLIB:=MEMLIB-AR ;
                             allera findepl
                       fin ;
       co la mémoire libre de la section numéro i est insuffisante ;
          le mot d'état de cette section doit être mis à jour co
          R:=-MEMLIB + 1er demi-mot de {PILDEPIL+1} , i ;
          MSP,R PILDEPIL,i ;
          co puis on passe à la section de pile suivante co
          i:=i+2 ;
          MEMLIB:=1er demi-mot de {PILDEPIL+1}, i + 2ème demi-mot
                  de {PILDEPIL+1}
          AL:=PILDEPIL,i - 2ème demi-mot de {PILDEPIL+1}, i + 1 ;
          si AL = ENTSEC alors MEMLIB:=MEMLIB - AR
                        sinon allera rang ;
```

```
findepl:  R:=-MEMLIB + 1er demi-mot de {PILDEPIL+1}, i ;
          nsec:=2ème demi-mot de {PILDEPIL+1} et X'7FFF' ; co
                nombre de sections co
          pour i:=i+2 pas 2 jusqua nsec faire
               R:=R - 2ème demi-mot de {PILDEPIL+1}, i ;
               MSP,R MEM ;
               PLW,R MEM ;
               PSW,R MEM ;
               co l'espace mémoire alloué est mis à jour co
               R:=AVECT ; R1:=T ;
               17[R]:=AR ;
               16[R]:=1,R1 + 2,R1 ;
               co construction du vecteur de renseignement de
                  ENTSEC co
fin :  fin
```

B I B L I O G R A P H I E

[1] F. BELLEGARDE
FACE. Langage d'écriture de compilateurs. Définition et implémentation.
Thèse de spécialité, Nancy (1972).

[2] C. PAIR
Cours de compilation.

[3] J. MAROLDT
Définition de FACE. Langage pour l'écriture des compilateurs.
Implémentation d'un sous-ensemble.
Thèse de docteur ingénieur, Nancy (1972).

[4] C.I.I.
SYMBØL-METASYMBØL. Manuel d'utilisation C900952.

[5] D. GRIES
Compiler construction for digital computers.
John Wiley and Sons. INC New-York (1971).

[6] COLMERAUER
Total precedence relations.
J. of A.C.M., 17 (1970)

[7] J.C. HELIARD
SYNPRØC. Manuel d'implémentation.
C.I.I. Carnet de notes (1971).

[8] ICHBIAH, MORSE
A technique for generating almost optimal Floyd Evans productions for precedence grammars.
Communications A.C.M., vol. 13 n° 8 (1970).

[9] KNUTH
On the translation of languages from left to right.
Information and Control, 8 (oct. 1965).

NOM DE L'ETUDIANT : PLATRE Hubert

NATURE DE LA THESE : Doctorat de Spécialité en Mathématiques Appliquées (Informatique)

VU, APPROUVE

& PERMIS D'IMPRIMER

NANCY, le 21 Février 1975

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

J.R. HELLUY

UNIVERSITÉ NANCY

E R R A T A

. SOMMAIRE Annexe 2

"AVANCCAR" au lieu de "AVANCARR"

. PAGE I.7. dans le schéma

"compilateur FACE" au lieu de "compilation FACE"

. PAGE II.22. 3ème alinéa

" $\varphi \xrightarrow{*} \Lambda$ (1)" au lieu de " $\varphi \xrightarrow{*} \Lambda$"

. PAGE II.24. 4ème ligne

"Soient t_1 et t_2" au lieu de "Soit t_1 et t_2"

. PAGE II.24. avant b)

"$c \in T \cap (J^*(B_1) \cup J^*(B_2) \ldots \cup J^*(B_n))$" au lieu de

"$c \in T \cap J^*(B_1) \cup J^*(B_2) \ldots \cup J^*(B_n)$

. PAGE II.25. 2ème ligne

"$\forall t \in T$" au lieu de "$\forall t \in R$"

. PAGE II.26. 1er ligne

"procédure lire" au lieu de "procédure lect"

. PAGE IV.2. dernier paragraphe

lire appel de WARSHALL ;

appel de VØISEVAL ;

appel de VERIF ;

. PAGE VI.23. Paragraphe 6.7.

"Le sous-programme VERIFA doit contrôler la grammaire en fonction des règles (D4), (D5) et vérifier le déterminisme des symboles terminaux"

au lieu de

"Le sous-programme VERIFA doit contrôler la grammaire en fonction des règles (D3), (D4) et (D5)".

. 3 lignes plus loin :

"règles (D4) et (D5)" au lieu de "règles (D3) et (D4)".

. ligne suivante :

"les symboles terminaux" au lieu de "la grammaire pour la règle (D5)".

. PAGE VI.31. 3ème ligne

"contrôle du déterminisme des symboles terminaux"

au lieu de

"contrôle de la grammaire en fonction de la règle D5".

Page VII.12. ajouter en bas

"Cette technique de la reprise assistée s'applique aussi bien à l'analyse descendante qu'à l'analyse ascendante ; mais elle est plus utile en ascendante, car avec la reprise automatique, la probabilité de sortie de messages d'erreurs multiples dûs à une cause unique y est plus importante qu'en analyse descendante (la reprise se fait seulement sur la relation vs, et non sur une règle en cours d'analyse)".

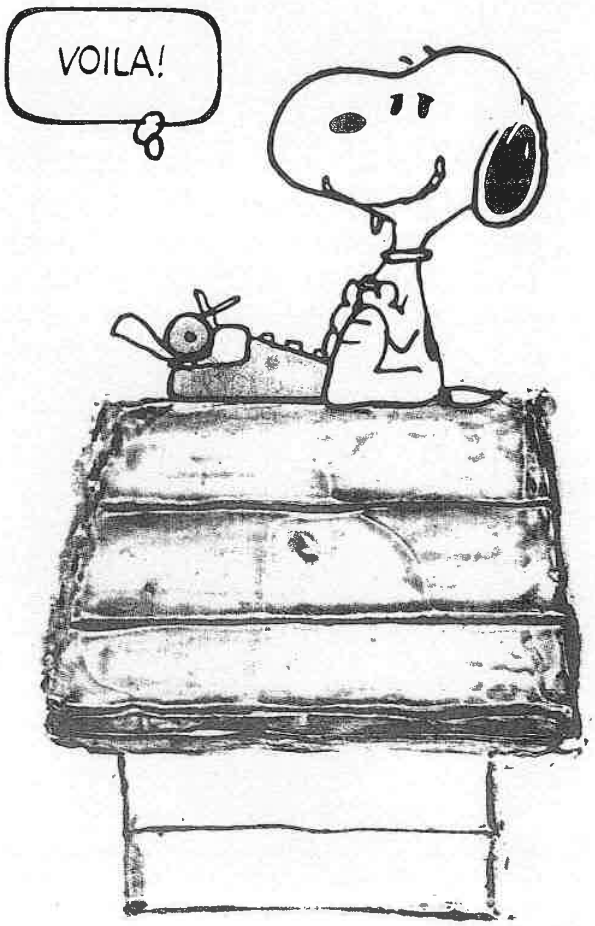
Page VIII.1. 8ème ligne du 3ème alinéa.

"Les règles (D3), (D4) et (D5)" au lieu de "les règles (D'3) et (D'4)".

Page VIII.2. 5ème ligne avant la fin

"(les indices des matrices de relations" au lieu de

"(les matrices indices des matrices de relations".


VOILA!