Méthodes d'analyses syntaxiques descendantes pour langages "context-free"

Xuan Nguyen-Dinh

N° d'ordre :

T H E S E

présentée à la

LA FACULTE DES SCIENCES DE L'UNIVERSITE DE

GRENOBLE

pour obtenir

LE TITRE DE DOCTEUR DE TROISIEME CYCLE

"MATHEMATIQUES APPLIQUEES"

-:-:-

par

NGUYEN-DINH Xuan

Licencié ès Sciences

-:-:-

METHODES D'ANALYSES SYNTAXIQUES DESCENDANTES POUR LANGAGES "CONTEXT-FREE"

-:-:-

Thèse soutenue le 21 décembre 1965 devant la Commission d'Examen :

MM. KUNTZMANN — Président

VAUQUOIS, GASTINEL, BOLLIET — Examinateurs

FACULTE DES SCIENCES

LISTE DES PROFESSEURS

DOYENS HONORAIRES M. FORTRAT P.

M. MORET L.

DOYEN M. WEIL L.

PROFESSEURS TITULAIRES

MM.	NEEL L.	MAGNETISME
	HEILMANN R.	CHIMIE ORGANIQUE
	KRAVTCHENKO J.	MECANIQUE RATIONNELLE
	CHABAUTY C.	MATHEMATIQUES PURES
	PARDE M.	POTAMOLOGIE
	BENOIT J.	RADIOELECTRICITE
	CHENE M.	CHIMIE PAPETIERE
	BESSON J.	ELECTROCHIMIE
	WEIL L.	THERMODYNAMIQUE
	FELICI N.	ELECTROSTATIQUE
	KUNTZMANN J.	MATHEMATIQUES APPLIQUEES
	BARBIER R.	GEOLOGIE APPLIQUEE
	SANTON L.	MECANIQUE DES FLUIDES
	OZENDA P.	BOTANIQUE
	FALLOT M.	PHYSIQUE INDUSTRIELLE
	GALVANI O.	MATHEMATIQUES
	MOUSSA A.	CHIMIE NUCLEAIRE ET RADIOACTIVITE
	TRAYNARD P.	CHIMIE GENERALE
	SOUTIF M.	PHYSIQUE GENERALE
	CRAYA A.	HYDRODYNAMIQUE
	REULOS R.	THEORIE DES CHAMPS
	AYANT Y.	PHYSIQUE APPROFONDIE
	GALISSOT F.	MATHEMATIQUES PURES
Mlle	LUTZ E.	MATHEMATIQUES GENERALES
MM.	BLAMBERT M.	MATHEMATIQUES
	BOUCHEZ R.	PHYSIQUE NUCLEAIRE
	LLIBOUTRY L.	GEOPHYSIQUE
	MICHEL R.	GEOLOGIE ET MINERALOGIE
	BONNIER E.	METALLURGIE
	DESSAUX G.	PHYSIOLOGIE ANIMALE
	PILLET E.	ELECTROTECHNIQUE
	DEBELMAS J.	GEOLOGIE GENERALE
	GERBER R.	MATHEMATIQUES PURES
	PAUTHENET R.	ELECTROTECHNIQUE
	VAUQUOIS B.	CALCUL ELECTRONIQUE
	SILBER R.	MECANIQUE DES FLUIDES
	MOUSSIEGT J.	ELECTRONIQUE
	BARBIER J.C.	PHYSIQUE EXPERIMENTALE

	BUYLE-BODIN M.	ELECTRONIQUE
	KOSZUL J.L.	MATHEMATIQUES
	DREYFUS B.	THERMODYNAMIQUE
	VAILLANT F.	ZOOLOGIE
	KLEIN J.	MATHEMATIQUES PURES
	SENGEL P.	ZOOLOGIE
	ARNAUD P.	CHIMIE
	BARJON R.	PHYSIQUE NUCLEAIRE
	BARNOUD F.	BIOSYNTHESE DE LA CELLULOSE

PROFESSEURS ASSOCIES

MM.	WAGNER H.	BOTANIQUE
	NAPP-ZINN K.	BOTANIQUE

PROFESSEURS SANS CHAIRE

Mme	KOFLER L.	BOTANIQUE
	DEPASSEL R.	MECANIQUE
	PERRET R.	SERVOMECANISME
Mme	BARBIER M.J.	ELECTROCHIMIE
	COHEN J.	PHYSIQUE
	GIDON P.	GEOLOGIE
Mme	SOUTIF J.	PHYSIQUE GENERALE
	GIRAUD P.	GEOLOGIE
	GASTINEL N.	MATHEMATIQUES APPLIQUEES
	LACAZE A.	THERMODYNAMIQUE
	GLENAT R.	CHIMIE ORGANIQUE
	BRISSONNEAU P.	PHYSIQUE GENERALE
	DUCROS P.	MINERALOGIE
	ANGLES D'AURIAC	MECANIQUE DES FLUIDES
	ROBERT A.	CHIMIE PAPETIERE
	COUMES A.	ELECTRONIQUE
	PEBAY-PEROULA	PHYSIQUE
	DEGRANGE C.	ZOOLOGIE
	GAGNAIRE D.	CHIMIE PAPETIERE
	RASSAT A.	CHIMIE
	PERRIAUX J.	GEOLOGIE
	BARRA J.	MATHEMATIQUES APPLIQUEES

PROFESSEURS HONORAIRES

MM.	FORTIER A.	MECANIQUE DES FLUIDES
	BRELOT M.	MATHEMATIQUES
	WOLFERS F.	PHYSIQUE
	DORIER A.	ZOOLOGIE

MAITRES DE CONFERENCES

MM.	BIAREZ J.	MECANIQUE DES FLUIDES
	DODU J.	MECANIQUE DES FLUIDES

	DOLIQUE J.M.	ELECTRONIQUE
	HACQUES G.	MATHEMATIQUES APPLIQUEES
	LANCIA R.	PHYSIQUE AUTOMATIQUE
	POULOUJADOFF M.	ELECTROTECHNIQUE
	KAHANE A.	PHYSIQUE
Mme	BONNIER J.	CHIMIE
Mme	KAHANE J.	PHYSIQUE
	DEPORTES C.	CHIMIE MINERALE
	DEPOMMIER P.	PHYSIQUE NUCLEAIRE
	CAUQUIS G.	CHIMIE GENERALE
	BONNET G.	PHYSIQUE
Mme	BOUCHE L.	MATHEMATIQUES
	COLOBERT L.	PHYSIOLOGIE ANIMALE
	PAYAN J.J.	MATHEMATIQUES
	CAUBET J.P.	MATHEMATIQUES
	LAURENT P.	MATHEMATIQUES APPLIQUEES
	BERTRANDIAS J.P.	MATHEMATIQUES APPLIQUEES
	BRIERE G.	PHYSIQUE
	LAJZEROWICZ J.	PHYSIQUE
	VALENTIN J.	PHYSIQUE
	DESRE P.	METALLURGIE
	BONNETAIN L.	CHIMIE MINERALE

MAITRE DE CONFERENCES ASSOCIE

MM.	RADELLI L.	GEOLOGIE

Je tiens à exprimer ma profonde reconnaissance à

Monsieur le Professeur KUNTZMANN, Directeur de l'Institut de Mathématiques Appliquées de Grenoble, qui a bien voulu me faire l'honneur de présider le Jury,

Monsieur le Professeur VAUQUOIS, Directeur du Centre d'Etudes pour la Traduction Automatique, qui a bien voulu faire partie du Jury,

Monsieur le Professeur GASTINEL, qui s'est intéressé à mon travail, pour ses encouragements bienveillants,

Monsieur BOLLIET, Ingénieur au C.N.R.S., qui a guidé mes recherches et qui a su les orienter vers les résultats les plus intéressants,

Monsieur COHEN, Attaché de Recherches au C.N.R.S., dont l'aide et les conseils judicieux m'ont permis de mener à bien ce travail.

Je remercie également tous les membres du Laboratoire de Calcul, dont l'aide me fut précieuse,

En particulier,

Monsieur TRILLING, Ingénieur au Laboratoire, dans la rédaction et la présentation de ce document,

Mademoiselle BRASSEUR, Maître Assistant à la Faculté des Sciences,

Monsieur DU MASLE , Ingénieur au C.N.R.S., pour leurs expériences profondes et leurs conseils bienveillants.

et,

Madame FRAIMBAULT, Mademoiselle TOURNOUD, Monsieur MOUNET, à qui je dois la réalisation matérielle de cet ouvrage.

A ma Mère,

A la mémoire de mon Père.

TABLE DES MATIERES

METHODES D'ANALYSES SYNTAXIQUES DESCENDANTES
POUR LANGAGES "CONTEXT-FREE"

-:-:-

INTRODUCTION

Le nombre de langages de programmation n'a cessé d'augmenter durant ces dernières années en raison des avantages qu'ils offrent pour la résolution des problèmes dans des domaines de plus en plus différents. Mais l'éventuelle diffusion d'un langage est étroitement liée au nombre de compilateurs opérationnels existant pour ce langage.

La construction d'un compilateur "hors série" (c. à.d. pour un langage donné) en langage machine est une opération longue et délicate et par conséquent coûteuse. La maintenance et la modification de tels compilateurs ne peuvent être confiées qu'à un groupe de spécialistes.

Les travaux actuels tendent vers une "paramétrisation" des compilateurs, c.à.d. :

- tout d'abord, vers la construction de compilateurs capables de traduire différents langages en un langage machine donné : le paramètre du compilateur est alors la description syntaxique et sémantique du langage.

- dans une étape ultérieure, vers l'élaboration de compilateurs du type précédent admettant comme paramètre supplémentaire la description du langage machine. Ces compilateurs seraient donc indépendants de la machine considérée.

Dans tous les cas, le processus de génération nécessite une analyse du langage source donnant lieu à la reconnaissance de sous-ensembles de symboles d'une chaîne d'entrée (le programme à traduire) qui ont un contenu sémantique suffisant pour provoquer le déclenchement d'une action du générateur. Dans les compilateurs usuels (jusqu'en 1960) les phases d'analyse et de génération étaient imbriquées et rendaient la paramétrisation particulièrement complexe. Dans quelques compilateurs produits après 1960 (cf. Ref. 1 et 2) la phase d'analyse est dissociée de celle de la génération et on observe l'importance que l'analyse syntaxique joue dans l'efficacité du compilateur. C'est ce fait qui a suscité l'attention de plusieurs chercheurs vers l'analyse syntaxique. La reconnaissance est pour la majorité des langages de programmation un procédé simple mais coûteux au point de vue du temps machine (en tenant compte de la vitesse et de l'organisation des calculateurs existants).

Les méthodes pour accélérer l'analyse ont pris une importance considérable et on peut dire actuellement que la construction pratique d'un compilateur paramétrisé dépend essentiellement de la méthode d'analyse syntaxique adoptée.

Les méthodes d'analyse des langages de programmation usuels se groupent communément en deux classes :

- les méthodes ascendantes (bottom up)

- les méthodes descendantes (top down)

L'objet de ce travail est l'étude des procédés principaux d'analyse descendante qui, par ailleurs, ont été parmi les premiers à être utilisés dans les compilateurs paramétrisés (cf.[Ref. 2]).

De récents travaux dans le domaine des langues naturelles ont montré qu'une transformation convenable des règles de syntaxe peut conduire à accélérer et simplifier le processus d'analyse.

C'est surtout dans cette voie que nous avons concentré notre effort. Une telle méthode de reconnaissance est décrite ici et a été appliquée dans l'analyse des langages algorithmiques du type d'ALGOL. Ce document décrit les résultats de cette investigation et donne une conclusion sur ses avantages et ses inconvévients pour son incorporation dans un compilateur paramétrisé.

La première partie de ce travail présente les aspects théoriques généraux de la question traitée. On y décrit aussi deux méthodes d'analyse descendante et on en profite pour donner un aperçu des problèmes posés par la génération.

D'autres aspects théoriques sont ensuite abordés : ceux ayant trait à la preuve d'équivalence entre certains types de descriptions syntaxiques. Seuls les énoncés des théorèmes démontrant cette équivalence sont présentés.

Sur ces bases, nous passons à la description de

l'algorithme qui a été développé pour effectuer une telle transformation.

L'algorithme d'analyse proprement dit est ensuite décrit dans les moindres détails et appliqué au langage ALGOL.

Enfin, l'utilisation pratique d'un tel algorithme a suggéré l'introduction d'artifices de programmation qui permettent d'accélérer considérablement le temps d'analyse. La description détaillée de tels artifices est aussi présentée dans ce travail.

Note. Tous les algorithmes présentés ont été testés sur l'ordinateur IBM 7044 de l'Institut de Mathématiques Appliquées de l'Université de Grenoble. On y dispose de compilateurs ALGOL et FORTRAN générant un programme objet en langage MAP (Macro-Assembly Program).

CHAPITRE I

ASPECTS THEORIQUES

I.1. DEFINITIONS PRELIMINAIRES(1). GRAMMAIRES DE CHOMSKY.

Tout d'abord essayons de décrire d'une façon intuitive le concept de "grammaire".

Un vocabulaire est un ensemble fini de primitives que nous ne définirons pas mais que nous appelerons symboles, lettres ou mots. Une chaîne consiste en un nombre fini de symboles liés par concaténation. Nous allons considérer un langage de programmation comme l'ensemble de toutes les chaînes définies par une "grammaire" sans attacher d'importance à leur "signification" ou partie sémantique.

Posé d'une façon plus rigoureuse, le problème essentiel est celui de définir un ensemble potentiellement infini de chaînes en utilisant un vocabulaire fini et un ensemble fini de règles. Ces règles appelées règles de syntaxe sont utilisées pour définir le langage.

Observons que pour spécifier un langage, on pourrait utiliser :

- soit un algorithme ou une machine produisant tous les éléments du langage et eux seuls.

(1) En écrivant cette partie nous nous sommes inspirés dans l'excellent travail de S. Greibach présenté en Ref. 3.

-soit un algorithme ou une machine reconnaissant tous les éléments du langage et eux seuls.

Cette dernière façon de procéder nécessite des règles de réécriture du type

$$\xi \longrightarrow \psi \qquad (1)$$

où ξ et ψ sont des chaînes. Quand cette règle est appliquée à la chaîne $\alpha\ \xi\ \beta$ où les chaînes α et β peuvent être simultanément ou non nulles, le résultat est une chaîne $\alpha\ \psi\ \beta$ et nous écrivons :

$$\alpha\ \xi\ \beta \longrightarrow \alpha\psi\beta$$

La flèche $\longrightarrow$ indique l'application d'une règle.

L'application d'une succession finie de règles est indiquée par $\overset{*}{\longrightarrow}$ c.à.d. $\alpha \overset{*}{\longrightarrow} \beta$ si et seulement s'il existe $\alpha = \alpha_0, \ldots, \alpha_n = \beta$ telles que $\alpha_i \longrightarrow \alpha_{i+1}$ pour $i = 0, 1, \ldots, n-1$. Nous dirons alors que β est générée à partir de α. Les règles du type (1) sont appelées <u>productions</u>.

Une grammaire de Chomsky G est un quadruplet $(V_N, V_T, S, \mathcal{R})$ où: V_T, vocabulaire terminal, est celui sur lequel sont formées les chaînes du langage correspondant à la grammaire.

V_N est un vocabulaire adjoint dit "non terminal" et tel que

$$V_N \cap V_T = \emptyset \text{ (ensemble vide)}$$

$S \in V_N$ est un élément distingué appelé axiome, à partir duquel toutes les chaînes du langage sont générées par l'application d'une succession finie de productions contenues dans l'ensemble $\mathfrak{R}$

Le langage correspondant à G est noté L(G).

Sauf indications contraires, dans tout ce qui suit, les lettres majuscules désignent des symboles non terminaux (ou métavariables), les lettres minuscules des symboles terminaux, les lettres grecques des chaînes.

Introduisons une série de restrictions sur la forme de réécriture.

La première restriction consiste à imposer qu'on ne puisse réécrire qu'un seul symbole à la fois. Les règles de réécriture s'appliquent seulement sur les éléments de V_N. Elle correspond aux grammaires de type 1.

Définition. Une grammaire est de type 1 ou "context sensitive" si les règles de $\mathfrak{R}$ sont de la forme :

$$\alpha \not\in \beta \longrightarrow \alpha \not\gamma \beta$$

La seconde restriction implique en plus, que le symbole traité puisse être réécrit indépendamment de son contexte : elle correspond aux grammaires de type 2.

Définition. Une grammaire est de type 2 ou "context free" si les règles de $\mathfrak{R}$ sont de la forme

$$Z \longrightarrow \gamma \qquad \text{où } \gamma \neq \Delta \text{ (chaîne vide)}$$

Ces grammaires sont appelées "context free" par opposition aux précédentes ; c'est d'ailleurs sur de telles grammaires que nous porterons notre attention. On sait, en effet, que la plupart des langages de programmation peuvent être définis syntaxiquement en utilisant ces grammaires "context-free". Le lecteur déjà familiarisé avec la forme normale de Backus (voir rapport ALGOL 60) pourra constater l'équivalence de celle-ci et de la définition précédente : la table ci-dessous donne la correspondance des notations.

Notation de la forme normale de Backus	Définition de grammaire (Chomsky)
::=	$\longrightarrow$
<métavariable>	éléments de V_N
symbole de base	éléments de V_T

La forme normale de Backus utilise aussi le symbole "|" qui permet de grouper les règles ayant la même partie gauche (élément qui précède le signe ::=)

Les règles de la forme

$$Z \longrightarrow Z\ \alpha$$

sont dites récursives à gauche. Celles de la forme :

$$Z \longrightarrow \alpha\ Z$$

sont dites récursives à droite.

Le sujet d'une règle est l'élément situé à gauche du signe $\longrightarrow$.

Enfin, pour compléter cette classification des grammaires de Chomsky, définissons-en le dernier type.

Définition. Une grammaire est de type 3 ou d'états finis si les règles de $\mathcal{R}$ sont de la forme :

$$Z \longrightarrow a\ Y$$

$$\text{ou } Z \longrightarrow a$$

I.2 LES DIFFERENTS TYPES D'ANALYSES SYNTAXIQUES.

Nous nous intéressons seulement à l'analyse des langages "context-free". Le problème est de déterminer si une chaîne w composée d'éléments de V_T peut être générée à partir de l'axiome, en référence à une grammaire "context free". Si à toute chaîne du langage on associe un formalisme mettant en évidence une suite de dérivations qui conduit depuis S à

cette chaîne ω , on dit que l'on a obtenu une "structure" de cette chaîne.

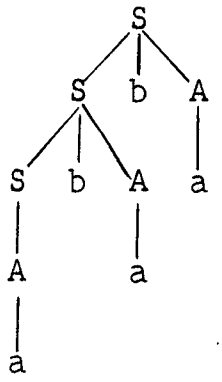
Exemple : Soit la grammaire G_1 définie par les règles de $\mathfrak{R}$ suivantes :

$$S \longrightarrow A$$
$$S \longrightarrow S\ b\ A$$
$$A \longrightarrow a$$

avec

$$V_T = \{a,\ b\}$$
$$V_N = \{S,\ A\}$$

Une structure de la chaîne a b a b a peut être représentée par l'arbre :



Parfois il existe plusieurs structures pour une même chaîne. On dit alors que le langage est "ambigu". Une méthode d'analyse tenant compte de ces ambiguïtés et recherchant toutes les structures possibles pour une chaîne donnée sera dite d'"analyse multiple" ; on parlera d'analyse unique si l'on se contente de déterminer une seule structure.

Une autre classification assez fréquente est celle d'analyse descendante et ascendante. Précisons, pour être clair,

le problème posé au calculateur. Celui-ci possède en mémoire un tableau contenant les règles de grammaire. Pour une chaîne donnée, supposons qu'il cherche une seule structure ; il veut donc retrouver une dérivation qui du symbole S conduit à la chaîne étudiée. Au départ, son but est d'obtenir S, sommet de l'arbre par lequel nous représentons cette structure. Mais ce but ne peut être atteint directement. Les noeuds de l'arbre deviendront des buts (1) intermédiaires qui seront placés sur une pile. Au moment de l'analyse du $i^{ème}$ symbole de la chaîne, cette pile contient un certain nombre d'éléments qui correspondent aux buts cherchés en partant des i premiers symboles. Le but qui se trouve au sommet de la pile est éliminé quand il a pu être obtenu à partir des éléments de la chaîne déjà lus. L'ordre dans lequel ces buts intermédiaires sont placés sur la pile, dépend du type d'analyse effectuée.

Si l'analyse est ascendante, on place d'abord sur la pile les buts correspondants aux noeuds de l'arbre les plus proches des symboles de la chaîne d'entrée.

Si l'analyse est descendante, les premiers buts placés sur la pile correspondent aux noeuds les plus proches de S.

Cette différence peut être exprimée d'une autre façon. Pour une grammaire présentée avec le formalisme proposé par TAYLOR, TURNER et WAYCHOFF et utilisé par la carte syntaxique d'ALGOL (Cf. [Ref.4]) l'analyse descendante conduit à

(1) Un but est un élément du vocabulaire non terminal (métavariable) que l'on essaie d'obtenir et qui sera un noeud de l'arbre si cette tentative réussit.

examiner (c.à.d. à essayer comme but) les métavariables en allant de haut en bas sur la carte. Dans l'analyse ascendante, les métavariables sont examinées de bas en haut.

Dans cette dernière classification, les analyses uniques et multiples peuvent être considérées comme des sous-classes de chacune de deux classes précédentes.

CHAPITRE II

DEUX METHODES D'ANALYSE DESCENDANTE

Nous présentons dans ce chapitre un algorithme d'analyse descendante inspiré par Cheatham [Ref.7] et qui est aisément abordable et une méthode dégagée d'un article de Leavenworth [Ref.8]. Ce dernier utilise les résultats de l'analyse syntaxique pour traduire une chaîne du langage considéré : ceci permettra de donner une idée des problèmes posés par la génération.

II.1. ALGORITHME D'ANALYSE DESCENDANTE DE CHEATHAM.

Les chaînes traitées appartiennent à un langage moins général que les langages "context-free" : la restriction dans la grammaire porte sur la récursivité des règles.

Présentation de la grammaire.

Elle est transformée en un ensemble d'arbres. Ainsi la grammaire G1 :

S ⟶ A B c
A ⟶ a
A ⟶ a B
B ⟶ a b

est mise sous la forme :

S ⟶ A B c

A ⟶ a ⇢ B

B ⟶ a b

Nous supposerons tout d'abord que la grammaire ne contient pas de règles récursives à gauche. Nous verrons plus loin comment les traiter à condition qu'elles soient d'une forme particulière.

Cet algorithme est décrit en ALGOL. Il eut été possible de l'écrire sous la forme d'une procédure récursive. Nous avons cependant préféré utiliser des piles : on sait que les effets de bord et les appels par nom et par valeur sont différemment interprétés par les compilateurs et nous pensons que l'exposé de l'algorithme n'y a rien perdu en clarté.

Voyons d'abord comment ranger les règles : elles sont stockées dans trois tableaux à une dimension ARBRE, PRECEDANT, ALTERNANT d'indice J. Le symbole ⟶ n'est pas recopié.

Les tableaux correspondants à la grammaire G1 sont les suivants :

J	PRECEDANT	ARBRE	ALTERNANT
1	* *	S	*
2	1	A	*
3	2	B	*
4	3	c	*
5		*	*

J	PRECEDANT	ARBRE	ALTERNANT
6	* *	A	*
7	6	a	*
8	7	*	9
9	7	B	*
10	9	*	*
11	* *	B	*
12	11	a	*
13	12	b	*
14	13	*	*

ARBRE contient les éléments de la grammaire, préalablement codés, notés dans l'ordre de lecture des règles. Les fins de règles sont repérées par la marque "*" représentée par ETOILE dans le programme ALGOL.

PRECEDANT[J] indique quel élément précède dans une règle donnée l'élément J considéré ; pour le sujet de chacune d'elles, PRECEDANT[J] est égal au symbole "* *" représenté par DEUX ETOILES.

Considérons la représentation de la grammaire comme un ensemble d'arbres dont les racines sont les sujets des règles. Du tronc commun peuvent partir plusieurs branches. A la naissance de chacune d'elles, ALTERNANT[J] indique où se trouve dans ARBRE, le début de la branche suivante. Il joue le même rôle que la flèche pointillée. Ainsi pour la grammaire G1, à la flèche pointillée correspond ALTERNANT [8] = 9 .

Il est nécessaire de posséder des informations d'une part sur la nature des éléments, d'autre part sur les indices J correspondants aux sujets des règles. L'élément du tableau booléen TERMINAL est faux s'il correspond à un symbole non terminal. Le tableau RACINE contient pour chaque sujet l'indice désiré. Ainsi pour la grammaire G1 :

Symbole	Racine	Terminal
S	1	faux
A	6	faux
B	11	faux
a	*	vrai
b	*	vrai
c	*	vrai

Les chaines terminales à analyser sont rangées dans le tableau CHAINE à une dimension. CHAINE[I] contient le code numérique attaché à l'élément terminal situé à la $i^{ème}$ position de la chaîne ; cette dernière est examinée de gauche à droite.

On utilise deux piles. L'une BUT, contient les buts recherchés, chacun suivi par deux indices (voir exemples suivants). L'autre TROUVE, renferme les sujets des règles utilisées, ce qui correspond aux buts obtenus, ainsi que d'autres éléments (voir exemples suivants) permettant de reconstituer la chaîne d'entrée.

Décrivons sur quelques exemples de chaînes satisfaisant à la grammaire G1 le déroulement de l'algorithme.

Exemple 1. Soit la chaîne : a a b c.

Initialement, BUT contient le but final S suivi des deux valeurs suivantes : I+1 = 1 et J = 1 . Le pointeur I de CHAINE repère le symbole précédant celui examiné "a". Le pointeur J d'ARBRE repère S.

Nous devons chercher si la règle donnant S, c.à.d :

S —>A B c (1)

peut s'appliquer.

A est un non-terminal qu'on ne peut trouver directement. Il est donc placé dans la pile BUT suivi des valeurs I+1 = 1 et J+1 = 2 (car ARBRE [2] = A).

Le nouveau but A ne peut être formé qu'à partir des règles :

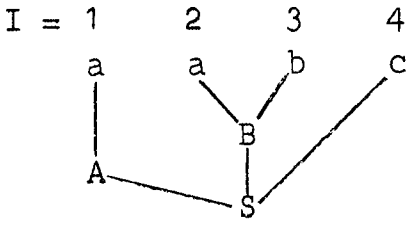
A —>a
ou A —>a B

L'algorithme essaye d'abord la première de ces règles. J prend la valeur de RACINE [A]= 6 ; c'est le niveau dans ARBRE du début des règles ayant A comme sujet. On compare ensuite ARBRE [J+1]= a à CHAINE [I+1] = a ; comme il lui est égal et que la règle est terminée (ARBRE [8] = ETOILE) on incrémente

I et J de 1 et on considère que A est "trouvé". A est déplacé de la pile BUT dans la pile TROUVE. Il y est suivi des deux valeurs précédemment rangées dans BUT et des valeurs courantes des pointeurs I et J.

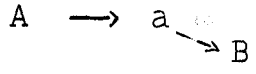
Il faut maintenant revenir à (1) ; J reprend sa valeur stockée dans BUT, J = 2. Et le processus continue, B est à son tour placé dans BUT et la règle B ⟶ a b essayée. Comme elle est satisfaite, B est transporté de BUT en TROUVE. I et J ont été incrémentés de 2 cette fois.

On examine ensuite le dernier élément de (1), il est bien identique à CHAINE[4]. Le but final S peut être alors transporté en TROUVE. Tous les éléments de la chaîne d'entrée ont été lus, l'analyse est correcte et correspond à la structure suivante (déduite immédiatement du contenu final de la pile TROUVE).

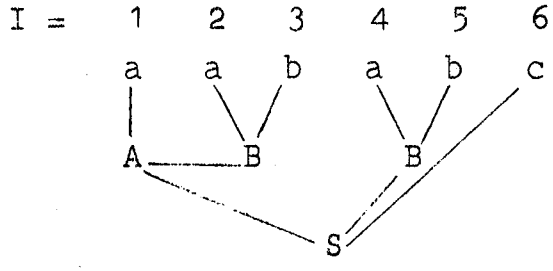


Exemple 2. Soit maintenant la chaîne : a a b a b c . Le début de l'analyse est semblable à celui décrit ci-dessus. A puis B sont placés dans TROUVE, mais on s'aperçoit que CHAINE[4] = a diffère du dernier élément c de (1). On est donc obligé de reculer et d'examiner s'il existe une nouvelle structure pour ce début de chaîne.

Nous recherchons d'abord si c = ARBRE [4] est à la naissance d'une branche c.à.d. si ALTERNANT [4] ≠ ETOILE. Ici, ce n'est pas le cas. Nous considérons alors l'élément B précédant c dans la règle (1). C'est un élément non terminal, il a donc été précédemment "trouvé". L'algorithme le replace dans la pile BUT et recherche s'il n'existe pas une autre règle que celle précédemment employée et fournissant cet élément non terminal B. I reprend la valeur qu'il avait quand on a trouvé B et J la valeur 13 correspondant au dernier élément ayant servi à former B (ces valeurs étaient stockées dans TROUVE). La recherche s'avérant ici infructueuse, B est enlevé de BUT, puis comme B = ARBRE [3] n'est pas à la naissance d'une branche, A est déplacé de TROUVE en BUT. Cette fois, on trouve une autre règle pouvant fournir A car ALTERNANT [8] = 9, c'est



On est conduit à empiler B sur BUT et le processus reprend comme au début. L'analyse correspond à la structure suivante :



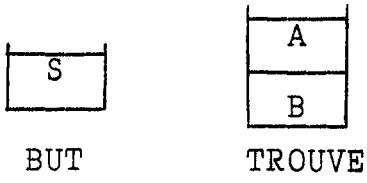
Notons qu'il faut lors d'un recul et lorsqu'on a "trouvé" plusieurs non-terminaux pour identifier un élément de la chaîne, enlever de TROUVE tous ces non-terminaux.

Exemple : Soit la grammaire G2 :

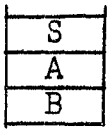
$$S \longrightarrow A\ c$$
$$A \longrightarrow B$$
$$A \longrightarrow ab$$
$$B \longrightarrow a$$

et soit la chaîne a b c à analyser.

Dans une première étape, on empile successivement dans BUT S, A, B et B, A passent dans TROUVE



A ce stade de l'analyse, le pointeur de CHAINE repère "b" qui n'est pas identique à "c" repéré par le pointeur d'ARBRE. Il faut revenir en arrière. A retourne dans BUT ainsi que B, sinon le contenu final de la pile TROUVE serait :



ce qui correspondrait à une structure erronnée de la chaîne. Signalons encore que l'algorithme tel qu'il est écrit en ALGOL s'arrête dès que les éléments de chaîne qu'il a examinés forment une chaîne syntaxiquement correcte. Si l'on désire qu'une chaîne soit complètement analysée, il faut l'encadrer

entre deux marqueurs " # " par exemple et considérer un nouvel axiome P correspondant à la règle unique

$$P \longrightarrow \# \; S \; \#$$

En ce qui concerne le traitement des règles récursives à gauche, on trouve dans [Ref.9] un algorithme les admettant si elles sont de la forme :

$$A \longrightarrow A \; a \; B$$

Le deuxième symbole de la partie droite doit être terminal. Enfin, nous verrons dans le chapitre III (page III.8) que de telles règles peuvent être transformées en règles récursives à droite ; ce qui résout le problème.

Disons encore qu'une étude préalable de la grammaire pourrait fournir des informations très intéressantes du point de vue de la rapidité d'analyse. Par exemple, il serait très utile de savoir si la métavariable considérée comme but peut effectivement représenter une chaîne commençant par le symbole terminal actuellement examiné. Dans le cas défavorable, le but serait abandonné. On comprend ainsi, que de nombreux essais infructueux seraient évités. Une telle étude préalable peut se faire aisément en s'inspirant de [Ref. 18]. L'algorithme présenté au chapitre III peut être considéré comme profitant d'une extraction de telles informations d'une grammaire avant d'aborder la phase d'analyse proprement dite.

Dans les pages suivantes figurent :

- l'algorithme écrit en ALGOL
- la grammaire GA que nous avons traitée, écrite sous forme normale de Backus. Elle décrit une partie du langage ALGOL, principalement les expressions arithmétiques et les expressions booléennes. Les symboles soulignés représentent des symboles que nous considérons comme terminaux.
- un exemple de chaîne (ou programme) satisfaisant à cette grammaire suivi du temps d'analyse observé. On le comparera avec ceux obtenus au chapitre IV.

ALGORITHME D'ANALYSE DESCENDANTE

(Inspiré de CHEATHAM)

```
   AXIØME := ARBRE[1];
   I := 0 ; J := K := L := 1 ;
   EMPILER BUT (AXIØME, I+1, J) ;
VERIFSITERM :
   si TERMINAL[ARBRE [J+1]] alors
   debut
        si CHAINE[I+1] = ARBRE[J+1] alors
        debut
             I := I+1 ; J := J+1 ;
VERIFINREGLE :
        si ARBRE[J+1] = ETØILE alors
        debut
             EMPILER TRØUVE(BUT[K-3] , BUT[K-2] ,BUT[K-1] ,I,J) ;
             si BUT[K-3] = AXIØME alors allera TERMINE ;
             J := BUT[K-1] ; K := K-3 ;
             allera VERIFIN REGLE
        fin ;
             allera VERIF  SITERM
        fin ;
        J := J+1 ; allera RECUL TERMINAL
   fin ;
   EMPILER BUT (ARBRE[J+1] , I+1 , J+1 ) ;
   J := RACINE [ARBRE[J+1]] ;
   allera VERIF SITERM ;

RECULER :
   si ¬ TERMINAL[ARBRE[J]] alors
   debut
       si PRECEDANT[J] = DEUX ETOILES alors
```

```
        debut
             I := BUT[K-2] -1 ;
             J := BUT[K-1] ;
             K := K-3 ; si K = 1 alors allera ERREUR ;
             RECHERCHE ALTERNANT (J)
        fin
    sinon
        debut
             EMPILER BUT (TRØUVE[L-5] ,TRØUVE[L-4] ,TRØUVE[L-3] ) ;
             I := TRØUVE[L-2] ;
             J := TRØUVE[L-1] +1 ;
             L := L-5
        fin ;
    allera RECULER
fin ;

RECUL TERMINAL :
    RECHERCHE ALTERNANT (J) ;
    allera RECULER ;
```

Les procédures suivantes sont utilisées dans cet algorithme:

```
procedure EMPILER BUT (T1, T2, T3) ; entier T1, T2, T3 ;
    debut entier V ;
          pour V := T1, T2, T3 faire
          debut
               BUT[K]:= V ;
               K := K+1
          fin
    fin EMPILER BUT ;
```

```
procedure EMPILER TRØUVE (T1, T2, T3, T4, T5);entier T1,T2,T3,T4,T5;
    debut entier V ;
          pour V := T1, T2, T3, T4 , T5 faire
          debut TRØUVE[L]:= V ;
                L := L+1
          fin
    fin EMPILER TRØUVE ;
```

```
procedure RECHERCHE ALTERNANT (N) ; entier N ;
    debut
          si ALTERNANT[N] ≠ ETØILE alors
          debut
               J := ALTERNANT[N]-1 ;
               allera VERIFINREGLE
          fin ;
          J := PRECEDANT[J] ;
          I := I-1
    fin RECHERCHE ALTERNANT ;
```

N.B. Quand on recherche si une fin de règle possède un alternant, on considère que le symbole "*" est terminal.
L'étiquette TERMINE indique une partie du programme signalant que la chaîne est correctement analysée.
L'étiquette ERREUR renvoie à une partie du programme qui signale l'erreur avant de s'arrêter.

GRAMMAIRE GA

< programme> ::= # < suite d'instructions> #
< suite d'instructions> ::= < instruction>|< instruction>;
< suite d'instructions>
< instruction> ::= < variable> := < expression arithmétique>

< expression arithmétique> ::=<expression arith. simple>|<prop.si>
<exp.arith.sple> sinon <expression arithmétique>
< expression arithm.sple> ::=<expr.arith.sple normale>|opadd
<expr. arith.sple normale>
< expression arithm.sple normale> ::=<terme>|<terme> opadd
<expr. arith. sple normale>
< terme> ::= <facteur>|<facteur> opmul <terme>
< facteur> ::= <primaire arithm>|<primaire arithm>↑<facteur>
< primaire arithmétique> ::=<variable>|nombre sans signe|(<ex-
pression arithmétique>)

< expression booléenne> ::=<expr.booléenne sple>|<prop si>
<exp.bool.sple> sinon <expr.bool.>
< expr.bool.simple>::=<implication>|<implication> ≡<expr.bool.sple>
< implication>::=<terme booléen>|<terme booléen>⊃<implication>
< terme booléen>::=<facteur booléen>|<facteur booléen>∨<terme bool>
< facteur booléen>::=<secondaire booléen>|<secondaire booléen> ∧
<facteur booléen>
< secondaire booléen> ::=<primaire booléen>|¬<primaire booléen>
< primaire booléen> ::=<variable>|valog |(<expr.booléenne>)|
<relation>

< relation> ::=<expr.arith.sple> oprel <expr.arith.sple>

<variable> ::= identificateur | identificateur[<liste d'indices>]

<liste d'indices> ::= <expr.arithm>|<expr.arithm> , <liste d'indices>

<proposition si> ::= si <expression booléenne> alors

Programme P1 :

```
#
D := si E > F  alors E sinon F ;
A[I] := B[I,J] * C[K x L,I, 9] ;
B[I+1,J]:= C [si I > J+K  alors L sinon  M,M↑3, N+7];
C[I,J,K * 3]:= si X ≤ Z ∧ (Y=T ∨ R)  alors -5 sinon A[-7]+B[M,-N];
A[I] := si X ∧ Y ∨ Z ≡ T ⊃ R ∧ ¬ S  alors  -B[I,J]↑9
                                     sinon   C[I,J,K] * (3+I)
#
```

Le temps d'analyse observé est de 4,3 s. Remarquons que la façon de présenter la grammaire joue un grand rôle quant à ce temps d'analyse. Ainsi, si on ne tient pas compte des éléments communs des règles, on obtient pour ce programme P1 le temps d'analyse exorbitant de 340,7 s.

II.2. ALGORITHME D'ANALYSE DESCENDANTE ABOUTISSANT A UNE GENERATION DANS UN LANGAGE OBJET.

Précisons tout d'abord qu'à la différence du précédent, cet algorithme ne s'écrit qu'en fonction d'une grammaire donnée, c.à.d. que la grammaire ne figure pas ici comme paramètre.

L'algorithme, pour la grammaire que nous avons traité en exemple, est décrit en FORTRAN IV et ceci a posé certains problèmes car ce langage de programmation n'admet pas de sous-programmes récursifs. Les chaînes du langage-source sont traduites dans un langage machine composé de macro-instructions de zéro à trois adresses :

Exemple : ADD OP1 OP2 RES

signifie affectation à RES de la somme des contenus de OP1 et OP2.

Nous montrerons d'abord comment se construit un algorithme et ensuite sur des exemples comment s'effectue la génération.

Description des grammaires admises.

Une grammaire est donnée sous une forme proche de la forme normale de Backus avec les deux modifications suivantes :

Les crochets "[" et "]" servent à grouper les parties droites ayant une partie commune.

Exemple : Une règle de la forme :

<A> ::= a <B> | a <C>

sera écrite sous la nouvelle forme :

<A> ::= a [<B> | <C>]

Les accolades "{" et "}" servent à décrire un nombre arbitraire d'occurences y compris l'occurence nulle.

Exemple : Une règle de la forme :

<A> ::= <A> b | a

qui donne :

<A> ::= a b^n | a (n > 0)

sera transformée en :

<A> ::= a{b}

De cette manière, on repère donc les règles récursives à gauche.

Structure de l'algorithme.

C'est un ensemble de fonctions logiques qui sont, soit des reconnaisseurs, soit des générateurs.

Il y a deux sortes de reconnaisseurs :

- reconnaisseurs de base
- reconnaisseurs de chaîne.

Les reconnaisseurs de base reconnaissent un seul symbole ou une classe de lettres qui forment un symbole de base en lettres. Les reconnaisseurs de chaîne reconnaissent une métavariable.

Les générateurs, à la suite d'une phase de reconnaissance effectuée par une suite de fonctions logiques du type précédent génèrent une séquence de macro-instructions correspondant à l'analyse résultante. Ces fonctions logiques ont toujours la valeur vrai.

Les chaînes du langage source sont examinées de gauche à droite. Le programme principal revient à l'exécution de la fonction logique <program> (correspondant à l'axiome) qui, à son tour et suivant la définition syntaxique de l'axiome, déclenche l'appel d'autres fonctions logiques.

Il s'agit bien là d'une analyse descendante.

Le langage FORTRAN a été utilisé essentiellement à cause de son processus "optimisé" d'évaluation des expressions logiques. En effet, les termes ou facteurs dont elles sont composées ne sont pas tous nécessairement calculés.

1. L'expression logique est examinée de gauche à droite.
2. Le premier terme logique vrai dans une union de termes donne la valeur vrai à l'union et les termes suivants ne sont pas évalués.

Exemple : B1 V B2 V ... V Bk V ... V Bn est vrai si Bk est vrai

3. Le premier facteur logique faux dans une intersection de facteurs donne la valeur faux à l'intersection et les autres facteurs ne sont pas évalués.

Exemple : B1 ∧ B2 ∧ ... ∧Bk∧ ... ∧Bn est faux si Bk est faux.

Les valeurs des pointeurs de la chaîne d'entrée et du sommet de la pile utilisée (cf.p.II-25) sont modifiées au cours de l'évaluation de la fonction logique par effet de bord.

On voit donc comment écrire un algorithme d'analyse en s'inspirant de très près des règles de grammaire.

Exemple : Soit la règle

<som> ::= <iden1> + <iden 2>

La fonction logique som associée à <som> est égale à l'intersection, dans l'ordre, du reconnaisseur de chaîne iden 1 associé à <iden 1>, du reconnaisseur de base PLUS identifiant l'opérateur + et du reconnaisseur de chaîne iden 2 associé à <iden 2>. Si a+b est une chaîne à analyser par cette règle, l'évaluation de l'expression logique som provoque l'examen de " a"par iden 1 ; si "a" est du type désiré, iden 1 prend la valeur vrai et fait avancer de 1 le pointeur de chaîne. Il y a bien effet de bord, iden 2 ne traite pas le même élément de chaîne que iden 1.

Chaque fois que l'on doit utiliser une règle récursive, on essaie de grouper le plus grand nombre d'éléments de la chaîne d'entrée sous le même symbole non terminal. A cet effet, on emploie la fonction logique ARBNO : elle recherche un nombre arbitraire d'occurrences y compris l'occurrence nulle. Elle peut se décrire sous forme ALGOL par l'expression suivante (le calcul des expressions booléennes est supposé "optimisé").

arbno := ¬ arg ∨ arbno (arg)

Si arg est vrai, ¬ arg est faux et on cherche à nouveau arbno (arg) jusqu'à ce que arg soit faux.

Exemple : De la règle :

<iden> ::= <letter> { <letter> | <number> }

on déduit l'expression de la fonction iden:

iden := letter ∧ arbno (letter ∨ number)

Un identificateur est ainsi défini par une lettre suivie ou non d'autres lettres ou chiffres.

Reconnaisseurs de chaîne.

Leur rôle est de déterminer si une unité syntaxique du programme donné correspond bien à la définition de la grammaire. Certaines règles sont récursives. Mais en FORTRAN IV, on ne peut écrire les fonctions logiques correspondantes sous forme récursive. Pour tourner cette difficulté, nous avons associé à chacune de ces fonctions une autre fonction,

ce qui permet d'effectuer des appels récursifs indirects.

Exemple : Dans le cas de la règle <A> ::= a <A> la fonction logique associée à celle correspondant à A est celle correspondant à B telle que :

<A> ::= a <B>
et <B> ::= <A>

De plus, des sous-programmes en langage d'assemblage (MAP) sont nécessaires pour contrôler les niveaux de récursivité.

Reconnaisseurs de base

Leur rôle est de déterminer si un symbole ou une suite de symboles appartient au vocabulaire terminal du langage source.

Exemple : Supposons que nous ayons à examiner le symbole de base begin écrit sous la forme 'BEGIN'.

Le reconnaisseur de base correspondant détecte la présence de l'apostrophe, puis forme l'ensemble de lettres que constitue ce symbole de base jusqu'à l'autre apostrophe. Une fois formé, on compare cet ensemble avec les éléments d'un tableau contenant les symboles de base composés de lettres.

Générateurs

A chaque générateur est associé un numéro N. Pour

bien montrer à quel instant leur mécanisme est déclanché, représentons-les par gN et plaçons-les judicieusement dans les règles.

Exemple : <iden> ::=<letter><g7>{<letter> <g8>|<number> <g9>}

Après avoir détecté la première lettre de l'identificateur à l'aide de la fonction <letter>, le générateur <g7> permet d'introduire cette lettre au sommet d'une pile d'identificateurs en le cadrant à gauche. Les générateurs <g8> et <g9> servent à placer respectivement la lettre ou le chiffre suivant à droite de la lettre déjà introduite par g7 (sur l'IBM 7044, par exemple, les identificateurs ne seraient composés que de 6 caractères).

Les accolades sont traitées par la fonction arbno (arg) décrite plus haut. On peut donc écrire sous une forme plus fonctionnelle :

identifier := letter $\wedge$ g7 $\wedge$ arbno (letter $\wedge$ g8 $\vee$ number $\wedge$ g9)

Si la fonction letter prend la valeur vrai alors seulement g7 est exécuté et ainsi de suite (rappelons que les fonctions g7, g8 et g9 ont toujours la valeur vrai).

Traitement d'une grammaire s'inspirant de celle d'ALGOL.

Une chaîne du langage (ou programme-source) satisfaisant à cette grammaire est constituée d'une instruction ou d'une suite d'instructions précédée par les déclarations de tous les identificateurs utilisés dans le programme. En

fait, cette grammaire ne définit qu'une partie d'ALGOL.

Les procédures, les tableaux et les expressions booléennes ne sont pas traités ; il n'existe qu'un déclarateur iden...

Nous la présentons sous la forme déjà utilisée dans le paragraphe précédent et précisant l'emplacement d'appel des générateurs: 18 de ceux-ci sont employés. Nous préciserons plus loin le rôle des principaux générateurs.

```
<program> ::= begin <declaration> <g17> <statement>
                  {;<statement>} end   <g18>
<declaration>::= iden <iden> <g15> {, <iden>  <g15>} ;
<statement>  ::= <iden> [=<g16> <g5> <expression> <g6>|
                         :<g10> <statement>] |
                 go to <iden> <g13>|
                 begin <statement>{; <statement>} end|
                 <conditional statement>
<unconditional statement> ::= <iden> =<g16> <g5> <expression><g6> |
                 go to <iden> <g13>|
                 begin <statement>{; <statement>}  end
<conditional statement> ::= if <iden> <g16> then <g12><uncond.stat>
                [else <g11>[<unconditional statement>|<condi.stat.>]
                     |ø   ] <g14>
                     (vide)
<expression> ::= [-<g2> | +] <expression>|
                 [<iden> <g16>|(<g3> <expression>)<g4>]<g1>
                 {<op> [<iden> <g16> | (<g3> <expression>)<g4>]|<g1>}
<iden>       ::=<letter> <g7> {<letter> <g8>|<number> <g9>}
```

L'algorithme exposé en FORTRAN IV correspondant à cette grammaire figure à la page II.28 à II.42. Pour avoir une idée de son fonctionnement, nous allons étudier quelques exemples.

Exemple 1. Soit le programme-source

```
begin iden A1, B1 ;
      A1 = B1
end
```

La fonction <program> détecte d'abord l'existence de begin, puis appelle le reconnaisseur de chaîne <déclaration>, celui-ci à l'aide du reconnaisseur de symbole de base distingue iden ; il forme selon le processus indiqué plus haut l'identificateur A1. Le générateur <g15> activé, créé la pseudo-instruction suivante dans le programme objet :

P Z E A1

Une fois l'identificateur B1 traité, le contrôle revient à <program> qui déclenche cette fois l'action du reconnaisseur de chaîne <statement>. Celui-ci reconnaît que la structure de ce qui suit est celle d'une instruction d'affectation. Il en évalue donc la partie droite à l'aide de la fonction <expression> et affecte la valeur trouvée à la partie gauche. Le rôle de <statement> est terminé. Et après avoir identifié end la fonction logique <program> prend la valeur vrai. Le programme source est correct et le programme généré est :

```
P Z E  A1   créé par <g15>:réservation d'une mémoire
            à A1
P Z E  B1   créé par <g15>:réservation d'une mémoire
            à B1
```

B G N créé par <g17>: début de programme
S T O B1 A1 créé par <g6> : stocker B1 dans A1
E N D créé par <g18>: fin de programme.

Pour évaluer les expressions, on aurait certainement pu détailler les règles s'y rapportant (distinguer les opérateurs multiplicatifs et additifs,..., etc). On a préféré ici employer la méthode de Bauer et Samuelson [Ref.10]. On utilise pour traduire les chaînes satisfaisant à cette grammaire les différents tableaux et piles suivants :

N V A R est la pile des identificateurs
N O P R est la pile des opérateurs
N R S et N R 1 sont des piles nécessaires respectivement à garder les adresses de retour dans les fonctions appelées par récursivité indirecte (cf. reconnaisseurs de chaîne) et à sauvegarder les informations attachées aux appels imbriqués de la fonction arbno.

I D L I S T est le tableau contenant les identificateurs déclarés au début du programme.
N G T et N G L sont des tableaux contenant respectivement la liste des mémoires auxiliaires et la liste des étiquettes disponibles.

Les priorités affectées aux opérateurs sont les suivantes :

Opérateurs	Priorité
(, =	0
) , ; , else, end	1
moins unitaire, +, -	2
* , /	3
↑	4

Traitons un cas un peu plus complexe, celui des instructions conditionnelles.

Exemple 2. Soit l'instruction :

if C then A = B else B = A ;

Les générateurs utilisés successivement sont :

g16 qui vérifie que l'identificateur précédemment reconnu et placé au sommet de la pile N V A R figure bien dans I D L I S T.

g12 qui génère le test sur C et augmente de 1 le pointeur de N G L. L'étiquette considérée dans N G L est placée dans N V A R.

g5 qui introduit l'opérateur "=" dans N O P R et g6 qui produit l'instruction correspondante (affectation).

g11 qui créé un transfert à l'étiquette indiquée par le pointeur de N G L et génère l'étiquette pointée dans NVAR. La première étiquette remplace la seconde dans NVAR.

Et après le traitement de B = A, g14 qui place dans le programme généré l'étiquette trouvée dans NVAR.

Le programme objet est le suivant :

TRF	C	GLO1	créé par g12 : transfert à GLO1 si C est faux
STØ	B	A	créé par g6
TRA	GLØ2		créé par g11 : transfert à GLO2
LAB	GLØ1		création de l'étiquette GLO1
STØ	A	B	créé par g6
LAB	GLØ2		créé par g14

En conclusion, signalons que l'utilisation de FORTRAN IV permet une analyse et une génération relativement rapide. Le temps machine nécessaire au passage de l'exemple de programme présenté à la page II-46 et II-47 est d'environ 4 secondes.

Un inconvénient majeur est qu'on doive écrire un programme pour chaque grammaire. Il serait sans doute possible d'élaborer un algorithme permettant de générer pour une grammaire donnée le programme d'analyse correspondant ; ce serait un tout autre problème en ce qui concerne la génération de code machine.

On trouve dans les pages suivantes l'algorithme écrit en FORTRAN IV, la liste des macro-instructions, une table donnant la signification des principaux identificateurs utilisés et les résultats obtenus par le traitement d'un exemple.

```
FJOB            3021,0044,0010 NGUYEN-DINH X.  FORTRAN IV SYNTAX LANGUAGE
FIBJOB FOSYLA   MAP,DECK
FIBFTC PROP     LIST
C       FORTRAN IV AS A SYNTAX LANGUAGE
        COMMON LTTR,NBR,IS,IPS,NVD1(16),KBLA(10),KOPR(10),KSBSE(10),
       1       KMAC(20),NGT(50),NGL(50),MACRO(10),NVD2(20),INPUT(100),
       2       NVAR(100),NOPR(100),IDLIST(100),NRS(300),NR1(300)
        LOGICAL PROGR
        READ(5,500)(KSBSE(I),I=2,8)
  500 FORMAT(7A6)
        WRITE(6,505)(KSBSE(I),I=2,8)
  505 FORMAT(7A8)
        READ(5,520)(KMAC(I),I=2,14)
  520 FORMAT(13A3)
        WRITE(6,525)(KMAC(I),I=2,14)
  525 FORMAT(13A8)
        READ(5,540)(NGT(I),I=2,19)
  540 FORMAT(18A4)
        WRITE(6,545)(NGT(I),I=2,19)
  545 FORMAT(1X,18A6)
        READ(5,540)(NGL(I),I=2,19)
        WRITE(6,545)(NGL(I),I=2,19)
        KBLA(2)=0
        KBLA(3)=48
        KBLA(4)=KBLA(3)+48*64
        KBLA(5)=KBLA(4)+48*64**2
        KBLA(6)=KBLA(5)+48*64**3
        KBLA(7)=KBLA(6)+48*64**4
        KOPR(2)=128+16
        KOPR(3)=128+32
        KOPR(4)=192+44
        KOPR(5)=192+49
        KOPR(6)=256+45
        NVAR=1
        NOPR=1
        IDLIST=1
        NRS=1
        NR1=1
        NGT=2
        NGL=2
        CALL LRCRT
        IF(.NOT.PROGR(1)) CALL ERROR(500)
        WRITE(6,590)
  590 FORMAT(1H0,36X,42H  NVAR  NOPRIDLIST   NRS   NR1   NGT   NGL)
        WRITE(6,600) NVAR(1),NOPR(1),IDLIST(1),NRS(1),NR1(1),NGT(1),NGL(1)
  600 FORMAT(1H0,36X,7I6)
        STOP
        END
```

```
FIBFTC LF10    LIST
      LOGICAL FUNCTION PROGR(N)
      LOGICAL SYBSE,DECLA,EXEC,STATE,ARBNO,LSTATE
      EXTERNAL LSTATE
      PROGR=SYBSE(2).AND.DECLA(1).AND.EXEC(17).AND.STATE(1)
     1      .AND.ARBNO(LSTATE).AND.SYBSE(8).AND.EXEC(18)
      RETURN
      END

FIBFTC LF15    LIST
      LOGICAL FUNCTION DECLA(N)
      LOGICAL SYBSE,IDEN,EXEC,ARBNO,LIDEN,SYM
      EXTERNAL LIDEN
      DECLA=SYBSE(3).AND.IDEN(1).AND.EXEC(15).AND.ARBNO(LIDEN)
     1      .AND.SYM(43).AND.SYM(43)
      RETURN
      END

FIBFTC LF17    LIST
      LOGICAL FUNCTION LIDEN(N)
      LOGICAL SYM,IDEN,EXEC
      LIDEN=SYM(59).AND.IDEN(1).AND.EXEC(15)
      RETURN
      END

FIBFTC LF20    LIST
      LOGICAL FUNCTION STATE(N)
      LOGICAL IDEN,SYM,EXEC,SAE,STATEP,SYBSE,ARBNO,LSTATE,COSTA
      EXTERNAL LSTATE
      CALL SAVE
      STATE=IDEN(1).AND.(SYM(11).AND.EXEC(16).AND.EXEC(5).AND.SAE(1)
     1      .AND.EXEC(6).OR.SYM(43).AND.EXEC(10).AND.STATEP(1))
     2      .OR.SYBSE(4).AND.IDEN(1).AND.EXEC(13)
     3      .OR.SYBSE(2).AND.STATEP(1).AND.ARBNO(LSTATE).AND.SYBSE(8)
     4      .OR.COSTA(1)
      CALL UNSAVE
      RETURN
      END

FIBFTC LF21    LIST
      LOGICAL FUNCTION STATEP(N)
      LOGICAL STATE
      CALL SAVE
      STATEP=STATE(1)
      CALL UNSAVE
      RETURN
      END

FIBFTC LF22    LIST
      LOGICAL FUNCTION LSTATE(N)
      LOGICAL SYM,STATE
      CALL SAVE
      LSTATE=SYM(43).AND.SYM(43).AND.STATE(1)
      CALL UNSAVE
      RETURN
      END
```

```
FIBFTC LF25    LIST
       LOGICAL FUNCTION UCSTA(N)
       LOGICAL IDEN,SYM,EXEC,SAE,SYBSE,STATE,ARBNO,LSTATE
       EXTERNAL LSTATE
       CALL SAVE
       UCSTA=IDEN(1).AND.SYM(11).AND.EXEC(16).AND.EXEC(5).AND.SAE(1)
      1      .AND.EXEC(6).OR.SYBSE(4).AND.IDEN(1).AND.EXEC(13)
      2      .OR.SYBSE(2).AND.STATE(1).AND.ARBNO(LSTATE).AND.SYBSE(8)
       CALL UNSAVE
       RETURN
       END

FIBFTC LF30    LIST
       LOGICAL FUNCTION COSTA(N)
       LOGICAL SYBSE,IDEN,EXEC,UCSTA,COSTAP,LV
       CALL SAVE
       LV=.TRUE.
       COSTA=SYBSE(5).AND.IDEN(1).AND.EXEC(16).AND.SYBSE(6).AND.EXEC(12)
      1      .AND.UCSTA(1).AND.
      2      (SYBSE(7).AND.EXEC(11).AND.(UCSTA(1).OR.COSTAP(1)).OR.LV)
      3      .AND.EXEC(14)
       CALL UNSAVE
       RETURN
       END

FIBFTC LF31    LIST
       LOGICAL FUNCTION COSTAP(N)
       LOGICAL COSTA
       CALL SAVE
       COSTAP=COSTA(1)
       CALL UNSAVE
       RETURN
       END

FIBFTC LF35    LIST
       LOGICAL FUNCTION SAE(N)
       LOGICAL SYM,EXEC,SAEP,IDEN,OPTERM,ARBNO
       EXTERNAL OPTERM
       CALL SAVE
       SAE=(SYM(32).AND.EXEC(2).OR.SYM(16)).AND.SAEP(1).OR.
      1    (IDEN(1).AND.EXEC(16).OR.SYM(60).AND.EXEC(3).AND.SAEP(1)
      2    .AND.SYM(28).AND.EXEC(4)).AND.EXEC(1).AND.ARBNO(OPTERM)
       CALL UNSAVE
       RETURN
       END                                                    .../...
```

```
          . . ./. . .
FIBFTC LF36    LIST
       LOGICAL FUNCTION SAEP(N)
       LOGICAL SAE
       CALL SAVE
       SAEP=SAE(1)
       CALL UNSAVE
       RETURN
       END

FIBFTC LF38    LIST
       LOGICAL FUNCTION OPTERM(N)
       LOGICAL OP,EXEC,IDEN,SAE,SYM
       CALL SAVE




       OPTERM=OP(1).AND.(IDEN(1).AND.EXEC(16).OR.SYM(60).AND.EXEC(3)
      1        .AND.SAE(1).AND.SYM(28).AND.EXEC(4)).AND.EXEC(1)
       CALL UNSAVE
       RETURN
       END
```

```
FIBFTC LF40    LIST
       LOGICAL FUNCTION ARBNO(ARG)
       LOGICAL ARBNOP,ARG
       CALL STAC
       ARBNO=.NOT.ARG(1).OR.ARBNOP(ARG)
       CALL UNSTAC
       RETURN
       END

FIBFTC LF41    LIST
       LOGICAL FUNCTION ARBNOP(ARG)
       LOGICAL ARBNO,ARG
       CALL RSAVE
       ARBNOP=ARBNO(ARG)
       CALL UNSAVE
       RETURN
       END

FIBFTC LF45    LIST
       LOGICAL FUNCTION LETNBR(N)
       LOGICAL LETTER,NUMBER,EXEC
       LETNBR=LETTER(1).AND.EXEC(8).OR.NUMBER(1).AND.EXEC(9)
       RETURN
       END

FIBFTC LF46    LIST
       LOGICAL FUNCTION LETSV(N)
       LOGICAL LETTER,EXEC
       LETSV=LETTER(1).AND.EXEC(8)
       RETURN
       END

FIBFTC LF50    LIST
       LOGICAL FUNCTION IDEN(N)
       COMMON NVD1(2),IS,IPS,NVD2(16),KBLA(10),NVD3(270),NVAR(100)
       LOGICAL LETTER,EXEC,ARBNO,LETNBR,LV
       EXTERNAL LETNBR
       LV=LETTER(1).AND.EXEC(7).AND.ARBNO(LETNBR)
       IDEN=LV
       IF(.NOT.LV) RETURN
       I=NVAR
       NVAR(I)=NVAR(I)+KBLA(IPS+2)
       IF(IS.EQ.0) RETURN
       NVAR(I)=-NVAR(I)
       RETURN
       END

FIBFTC LF55    LIST
       LOGICAL FUNCTION LETTER(N)
       COMMON LTTR,NVD1(199),INPUT(100)
       LETTER=.FALSE.
       I=INPUT
       I=INPUT(I)
       IF(I.GT.16.AND.I.LT.26) GOTO 10
       IF(I.GT.32.AND.I.LT.42) GOTO 10
       IF(I.GT.49.AND.I.LT.58) GOTO 10
       RETURN
    10 LTTR=I
       CALL NOBLA
       LETTER=.TRUE.
       RETURN
```

```
FIBFTC LF56    LIST
       LOGICAL FUNCTION NUMBER(N)
       COMMON NVD1,NBR,NVD2(198),INPUT(100)
       NUMBER=.FALSE.
       I=INPUT
       I=INPUT(I)
       IF(I.GT.9) RETURN
       NBR=I
       CALL NOBLA
       NUMBER=.TRUE.
       RETURN
       END
FIBFTC LF57    LIST
       LOGICAL FUNCTION SYM(N)
       COMMON NVD1(200),INPUT(100)
       SYM=.FALSE.
       I=INPUT
       IF(INPUT(I).NE.N) RETURN
       CALL NOBLA
       SYM=.TRUE.
       RETURN
       END
FIBFTC LF58    LIST
       LOGICAL FUNCTION SYBSE(N)
       COMMON NVD1(2),IS,IPS,NVD2(16),KBLA(10),NVD3(10),KSBSE(10)
      1        NVD4(150),INPUT(100),NVAR(100)
       LOGICAL SYM,LETTER,EXEC,ARBNO,LETSV,LV
       EXTERNAL LETSV
       SYBSE=.FALSE.
       IN=INPUT
       LV=SYM(12).AND.LETTER(1).AND.EXEC(7).AND.ARBNO(LETSV)
       I=INPUT
       LV=LV.AND.INPUT(I).EQ.12
       IF(.NOT.LV) RETURN
       I=NVAR
       KS=NVAR(I)+KBLA(IPS+2)
       NVAR=NVAR-1
       IF(IS.EQ.0) GOTO 10
       KS=-KS
    10 IF(KS.EQ.KSBSE(N)) GOTO 20
       INPUT=IN
       RETURN
    20 CALL NOBLA
       SYBSE=.TRUE.
       RETURN
       END
```

```
FIBFTC LF60    LIST
       LOGICAL FUNCTION OP(N)
       COMMON NVD1(200),INPUT(100),NVD2(100),NOPR(100)
       NOPR=NOPR+1
       J=NOPR
       IN=INPUT
       I=INPUT(IN)
       IF(I.EQ.16.OR.I.EQ.32) GOTO 10
       IF(I.EQ.49) GOTO 20
       IF(I.NE.44) GOTO 40
       IF(INPUT(IN+1).NE.44) GOTO 20
       INPUT=INPUT+1
       NOPR(J)=256+45
       GOTO 30
    10 NOPR(J)=128+I
       GOTO 30
    20 NOPR(J)=192+I
    30 CALL NOBLA
       OP=.TRUE.
       RETURN
    40 NOPR=NOPR-1
       OP=.FALSE.
       RETURN
       END
```

```
:IBFTC LF80    LIST
      LOGICAL FUNCTION EXEC(N)
      COMMON LTTR,NBR,IS,IPS,NVD1(26),KOPR(10),NVD2(10),KMAC(20),
     1       NGT(50),NGL(50),MACRO(10),NVD3(120),NVAR(100),NOPR(100),
     2       IDLIST(100)
  600 FORMAT(1H0,4A8)
      CALL RSAVE
      EXEC=.TRUE.
      GOTO(10,20,30,40,50,60,70,80,90,100,110,120,130,140,150,
     1                                   160,170,180),N
   10 CALL PRIOR(K)
   12 I=NOPR
      I=NOPR(I)
      IF(I/64.LT.K) GOTO 1000
      IF(I.NE.128+31) GOTO 14
      MACRO(2)=KMAC(7)
      NOPR=NOPR-1
      I=NVAR
      MACRO(3)=NVAR(I)
      J=NGT
      MACRO(4)=NGT(J)
      NVAR(I)=NGT(J)
      NGT=NGT+1
      WRITE(6,600) MACRO(2),MACRO(3),MACRO(4)
      GOTO 12
   14 DO 16 J=2,6
      IF(I.EQ.KOPR(J)) GOTO 18
   16 CONTINUE
      CALL ERROR(20)
   18 MACRO(2)=KMAC(J)
      NOPR=NOPR-1
      I=NVAR
      MACRO(3)=NVAR(I-1)
      MACRO(4)=NVAR(I)
      J=NGT
      MACRO(5)=NGT(J)
      NVAR(I-1)=NGT(J)
      NVAR=NVAR-1
      NGT=NGT+1
      WRITE(6,600) MACRO(2),MACRO(3),MACRO(4),MACRO(5)
      GOTO 12
   20 NOPR=NOPR+1
      I=NOPR
      NOPR(I)=128+31
      GOTO 1000
   30 NOPR=NOPR+1
      I=NOPR
      NOPR(I)=60
      GOTO 1000
   40 NOPR=NOPR-1
      GOTO 1000
   50 NOPR=NOPR+1
      I=NOPR
      NOPR(I)=11
      NGT=2
      GOTO 1000
```

```
 60 MACRO(2)=KMAC(8)
    NOPR=NOPR-1
    I=NVAR
    MACRO(3)=NVAR(I)
    MACRO(4)=NVAR(I-1)
    NVAR=NVAR-2
    GOTO 940
 70 NVAR=NVAR+1
    I=NVAR
    IS=0
    IF(LTTR.LT.32) GOTO 72
    LTTR=LTTR-32
    IS=1
 72 NVAR(I)=LTTR*64**5
 *  IPS=5
    GOTO 1000
 80 I=NVAR
    IPS=IPS-1
    NVAR(I)=NVAR(I)+LTTR*64**IPS
    GOTO 1000
 90 I=NVAR
    IPS=IPS-1
    NVAR(I)=NVAR(I)+NBR*64**IPS
    GOTO 1000
100 MACRO(2)=KMAC(11)
    I=NVAR
    MACRO(3)=NVAR(I)
    NVAR=NVAR-1
    GOTO 920
110 MACRO(2)=KMAC(10)
    J=NGL
    MACRO(3)=NGL(J)
    WRITE(6,600) MACRO(2),MACRO(3)
    MACRO(2)=KMAC(11)
    I=NVAR
    MACRO(3)=NVAR(I)
    NVAR(I)=NGL(J)
    NGL=NGL+1
    GOTO 920
120 MACRO(2)=KMAC(9)
    I=NVAR
    MACRO(3)=NVAR(I)
    J=NGL
    MACRO(4)=NGL(J)
    NVAR(I)=NGL(J)
    NGL=NGL+1
    GOTO 940
130 MACRO(2)=KMAC(10)
    I=NVAR
    MACRO(3)=NVAR(I)
    NVAR=NVAR-1
    GOTO 920
140 MACRO(2)=KMAC(11)
    I=NVAR
    MACRO(3)=NVAR(I)
    NVAR=NVAR-1
    GOTO 920
```

```
150 I=NVAR
    IDLIST=IDLIST+1
    J=IDLIST
    IDLIST(J)=NVAR(I)
    MACRO(2)=KMAC(12)
    MACRO(3)=NVAR(I)
    NVAR=NVAR-1
    GOTO 920
160 I=NVAR
    IN=NVAR(I)
    J=IDLIST
    DO 162 I=2,J
    IF(IN.EQ.IDLIST(I)) GOTO 1000
162 CONTINUE
    CALL ERROR(10)
170 MACRO(2)=KMAC(13)
 *  GOTO 900
180 MACRO(2)=KMAC(14)
900 WRITE(6,600) MACRO(2)
    GOTO 1000
920 WRITE(6,600) MACRO(2),MACRO(3)
    GOTO 1000
940 WRITE(6,600) MACRO(2),MACRO(3),MACRO(4)
1000 CALL UNSAVE
    RETURN
    END
```

```
FIBFTC SR10    LIST
      SUBROUTINE LRCRT
      COMMON NVD1(200),INPUT(100)
      READ(5,500)(INPUT(I),I=2,73)
  500 FORMAT(72A1)
      WRITE(6,600)(INPUT(I),I=2,73)
  600 FORMAT(1H0,36X,72A1)
      IDC=64**5
      DO 20 I=2,73
      J=INPUT(I)
      IF(J.GE.0) GOTO 10
      INPUT(I)=IABS(J)/IDC+32
      GOTO 20
   10 INPUT(I)=J/IDC
   20 CONTINUE
      I=2
   30 IF(INPUT(I).NE.48) GOTO 40
      I=I+1
      GOTO 30
   40 INPUT=I
      RETURN
      END

FIBFTC SR15    LIST
      SUBROUTINE NOBLA
      COMMON NVD1(200),INPUT(100)
   10 INPUT=INPUT+1
      IF(INPUT.GT.73) GOTO 20
      I=INPUT
      IF(INPUT(I).NE.48) RETURN
      GOTO 10
   20 CALL LRCRT
      RETURN
      END
```

```
FIBFTC SR20    LIST
      SUBROUTINE PRIOR(K)
      COMMON NVD1(3),IPS,NVD2(16),KBLA(10),NVD3(10),KSBSE(10),
     1       NVD4(150),INPUT(100),NVAR(100)
      LOGICAL SYM,LETTER,EXEC,ARBNO,LETSV,LV
      EXTERNAL LETSV
      IN=INPUT
      I=INPUT(IN)
      J=INPUT(IN+1)
      IF(I.EQ.16.OR.I.EQ.32) GOTO 10
      IF(I.EQ.49) GOTO 20
      IF(I.NE.44) GOTO 30
      IF(J.NE.44) GOTO 20
      K=4
      RETURN
   10 K=2
      RETURN
   20 K=3
      RETURN
   30 IF(I.EQ.60) CALL ERROR(50)
      IF(I.EQ.28.OR.I.EQ.43.AND.J.EQ.43) GOTO 50
      LV=SYM(12).AND.LETTER(1).AND.EXEC(7).AND.ARBNO(LETSV)
      I=INPUT
      LV=LV.AND.INPUT(I).EQ.12
      IF(.NOT.LV) CALL ERROR(30)
      I=NVAR
      KS=NVAR(I)+KBLA(IPS+2)
      NVAR=NVAR-1
      IF(KS.EQ.KSBSE(8).OR.KS.EQ.KSBSE(7)) GOTO 40
      CALL ERROR(40)
   40 INPUT=IN
   50 K=1
      RETURN
      END

FIBFTC SR25    LIST
      SUBROUTINE ERROR(N)
      WRITE(6,600) N
  600 FORMAT(8H  ERROR ,I5)
      STOP
      RETURN
      END
```

```
FIBMAP  MS10
        PMC     ON
NRS     EQU     SCOMM+600
        USE     //
        CONTRL  //
SCOMM   BSS     900
        USE
        ENTRY   SAVE
SAVE    SAVE    1,2,4
        LAC     NRS,1
        TXI     *+1,4,9
        PXA     ,4
        STA     NRS,1
        SXD     E2,4
        TXI     *+1,4,6
E1      TXI     *+1,1,-1
        TXI     *+1,4,-1
        CLA     ,4
        STO     NRS,1
E2      TXH     E1,4,**
        CLA     NRS
        ADD     =7
        STO     NRS
        RETURN  SAVE
        END

FIBMAP  MS11
        PMC     ON
NRS     EQU     SCOMM+600
        USE     //
        CONTRL  //
SCOMM   BSS     900
        USE
        ENTRY   UNSAVE
UNSAVE  SAVE    1,2,4
        CLA     NRS
        SUB     =7
        STO     NRS
        LAC     NRS,1
        CLA     NRS,1
        PAX     ,4
        SXD     E2,4
        TXI     *+1,4,6
E1      TXI     *+1,1,-1
        TXI     *+1,4,-1
        CLA     NRS,1
        STO     ,4
E2      TXH     E1,4,**
        RETURN  UNSAVE
        END
```

```
FIBMAP  MS15
        PMC     ON
NRS     EQU     SCOMM+600
        USE     //
        CONTRL  //
SCOMM   BSS     900
        USE
        ENTRY   RSAVE
RSAVE   SAVE    1,2,4
        LAC     NRS,1
        TXI     *+1,4,11
        PXA     ,4
        STA     NRS,1
        SXD     E2,4
        TXI     *+1,4,6
E1      TXI     *+1,1,-1
        TXI     *+1,4,-1
        CLA     ,4
        STO     NRS,1
E2      TXH     E1,4,**
        CLA     NRS
        ADD     =7
        STO     NRS
        RETURN  RSAVE
        END

FIBMAP  MS20
        PMC     ON
NR1     EQU     SCOMM+900
        USE     //
        CONTRL  //
SCOMM   BSS     1200
        USE
        ENTRY   STAC
STAC    SAVE    1,2,4
        LAC     NR1,1
        TXI     *+1,4,12
        PXA     ,4
        STA     NR1,1
        SXD     E2,4
        TXI     *+1,4,6
E1      TXI     *+1,1,-1
        TXI     *+1,4,-1
        CLA     ,4
        STO     NR1,1
E2      TXH     E1,4,**
        TXI     *+1,4,-11
        CLA     ,4
        STO     NR1+1,1
        PAC     ,4
        CLA     ,4
        STA     NR1+2,1
        CLA     NR1
        ADD     =9
        STO     NR1
        RETURN  STAC
        END
```

```
FIBMAP  MS21
        PMC     ON
NR1     EQU     SCOMM+900
        USE     //
        CONTRL  //
SCOMM   BSS     1200
        USE
        ENTRY   UNSTAC
UNSTAC  SAVE    1,2,4
        CLA     NR1
        SUB     =9
        STO     NR1
        LAC     NR1,1
        CLA     NR1,1
        PAX     ,4
        SXD     E2,4
        TXI     *+1,4,6
E1      TXI     *+1,1,-1
        TXI     *+1,4,-1
        CLA     NR1,1
        STO     ,4
E2      TXH     E1,4,**
        CLA     NR1
        SUB     =1
        TZE     E3
        TXI     *+1,1,9
        CLA     NR1+1,1
        STO     *+2
        CLA     NR1+2,1
        BSS     1
E3      RETURN  UNSTAC
        END
```

LISTE DES MACRO-INSTRUCTIONS

ADD	P1	P2	P3	Additionner P1 à P2 et envoyer en P3 P1 + P2 ⟶ P3
SUB	P1	P2	P3	Soustraire P2 de P1 et envoyer en P3 P1 - P2 ⟶ P3
MPY	P1	P2	P3	Multiplier P1 par P2 et envoyer en P3 P1 x P2 ⟶ P3
EXP	P1	P2	P3	Elever P1 à la puissance P2 et envoyer en P3 P1 ↑ P2 ⟶ P3
UMN	P1	P2		Changer le signe de P1 et envoyer en P2 -P1 ⟶ P2
STO	P1	P2		Envoyer P1 en P2 P1 ⟶ P2
TRF	P1	P2		Si P1 est faux transférer à P2
TRA	P1			Transférer à l'étiquette P1
LAB	P1			Produire l'étiquette P1
PZE	P1			Réserver une mémoire à P1
BGN				Début de programme
END				Fin de programme

Signification des principales fonctions logiques et sous-programmes

Reconnaisseurs de chaînes.

PROGR(N)	représente	<program>.N : paramètre sans signification (obligation en FORTRAN)
DECLA(N)	"	<declaration>
LIDEN(N)	"	, <iden> <g15>
STATE(N)	"	<statement>
STATEP(N)	"	la fonction associée à STATE(N) pour la récursivité indirecte.
LSTATE(N)	"	; <statement>
UCSTA(N)	"	<unconditional statement>
COSTA(N)	"	<conditional statement>
COSTAP(N)	"	la fonction associée à COSTA(N)
SAE(N)	"	<expression>
SAEP(N)	"	la fonction associée à SAE(N)
OPTERM(N)	"	<op> [<iden><g16>|(<g3><expression>) <g4>] <g1>
IDEN(N)	"	<iden>

Reconnaisseurs de base.

LETNBR(N)	"	<letter><g8> |<number> <g9>
LETSV(N)	"	<letter> <g8>
LETTER(N)	"	<letter> : recherche d'une lettre
NUMBER(N)	"	<number> : recherche d'un nombre
SYM(N)		recherche d'un symbole en code décimal N
SYBSE(N)		" " " de base en lettre, n° d'ordre N
OP(N)		recherche d'un opérateur et affectation de priorité.

Générateurs.

EXEC(N)	générateurs avec N=1,...,18 représentant les différents gN.

Sous-programmes en FORTRAN.

LRCRT	Lire une carte et répartir les données en 72 mémoires.
NOBLA	Recherche d'un caractère non blanc.
PRIOR(K)	Affectation de la priorité de l'opérateur à K.
ERROR(N)	Ecrire ERROR N et arrêter en cas d'erreur

Sous-programmes MAP de récursivité.

SAVE	utilisé à l'entrée des fonctionsrécursives sans paramètre à sauvegarder.
UNSAVE	utilisé à la sortie des fonctions récursives sans paramètre à sauvegarder.
RSAVE	utilisé à l'entrée de ARBNOP(ARG) (fonction associée à ARBNO): argument à sauvegarder une fois.
STAC	utilisé à l'entrée de ARBNO (ARG) : argument à sauvegarder deux fois.
UNSTAC	utilisé à la sortie de ARBNO (ARG) : argument à sauvegarder deux fois.

CHAPITRE III

ANALYSE PREDICTIVE DE KUNO-OETTINGER

Nous exposons en premier lieu, dans ce chapitre, les bases théoriques sur lesquelles nous nous appuyons pour transformer une grammaire "context-free" en une grammaire sous forme standard. En effet, l'algorithme d'analyse descendante inspiré de Kuno-Oettinger [Ref.11] que nous décrivons ensuite n'opère que si la grammaire est de ce type.

Le processus d'analyse en est certainement simplifié ; nous verrons qu'il en est aussi accéléré. Enfin, notre attention s'est portée sur le problème de l'ordre de consultation des règles et nous pensons qu'il est particulièrement aisé d'incorporer à cet algorithme une technique de réordonnancement des règles.

III.1. THEOREMES D'EQUIVALENCE ENTRE UNE GRAMMAIRE CONTEXT-FREE ET UNE GRAMMAIRE SOUS FORME STANDARD.

Nous nous contenterons dans ce paragraphe d'énoncer les théorèmes employés pour aboutir à cette équivalence et de décrire, sans entrer dans les détails, le processus de la démonstration. On trouve les démonstrations des théoèremes dans [13,14].

Considérons une grammaire context-free G^* définie par le quadruplet $(V_N, V_T, S, \mathcal{R})$. Nous dirons qu'une grammaire G^* est sous forme standard si et seulement si toutes les règles de $\mathcal{R}$ sont de la forme :

pour tout Z

$$Z \longrightarrow a$$
$$Z \longrightarrow a\ Y_1 \ \ldots\ Y_n \qquad n \geqslant 1$$

Avant la présentation de ces théorèmes, donnons quelques définitions et précisons nos notations.

Soit V un vocabulaire fini (ici les lettres grecques sont des chaînes sur V),

- si $a \in V$, a désigne la chaîne formée du seul symbole a
- R + Q désigne α si et seulement si R ou Q désignent α simultanément ou non.
- R Q désigne $\alpha\,\beta$ si et seulement si R désigne α et Q désigne β.
- si R_i désigne α_i $(1 \leqslant i \leqslant n)$, $cl(R_1,\ldots,R_n)$ désigne une séquence $\varphi_1 \varphi_2 \ldots \varphi_m$ telle que $\varphi_k \in \{\Delta, \alpha_i \mid i=1,2,..,n\}$. On peut avoir $m > n$ et répétition d'un nombre quelconque de φ pris dans les α_i, l'ordre de concaténation étant indifférent.

Ex : cl(a) représente un élément quelconque de $\{a^n \mid n \geqslant 0\}$.
cl(a,b) représente un élément quelconque du monoïde (1) libre défini sur $\{a,b\}$.

Définissons maintenant une expression régulière :

- Δ, $\emptyset$, $a \in V$ sont respectivement des expressions régulières.

(1) Nous dirons que le monoïde libre sur un vocabulaire V est l'ensemble des chaînes pouvant être formées sur V.

-si R et Q sont des expressions régulières, R+Q et RQ le sont aussi.
-si Ri ($1 \leqslant i \leqslant n$) est une expression régulière, $cl(R_1, ..., R_n)$ l'est aussi.

On se ramène d'abord à de nouveaux types de grammaire se rapprochant plus de la forme standard.

Définition 1. Une grammaire à structure de phrase (phrase structure grammar) ou p s g est une grammaire $G_1 = (V_N, V_T, S, \mathcal{R})$ dont les règles sont de la forme :

$$Y \longrightarrow A_1 \ldots A_n \qquad n \geqslant 2 \qquad A_i \in V_T \cup V_N$$
$$Y \longrightarrow a$$

Une telle structure exclut :

- toute règle telle que $Y \longrightarrow \Delta$
- toute règle telle que $Y \longrightarrow Z$. On interdit ainsi d'éventuelles dérivations $Y \overset{*}{\longrightarrow} Y$.

On a pu démontrer :

Théorème : Pour toute grammaire G context-free, il existe une p s g générant $L(G) - \Delta$.

Le fait d'exclure la chaîne vide du langage correspondant à une p s g ne constitue pas une restriction importante. On peut donc considérer qu'il existe toujours une p s g équivalente à une grammaire context-free donnée.

```
          16  HEURES  55  MINUTES 39  SECONDES
 BEGIN    IDEN     GOTO     IF       THEN     ELSE     END
 ADD      SUB      MPY      DIV      EXP      UMN      STO      TRF      TRA      LAB      PZE      BGN      END
GT01  GT02  GT03  GT04  GT05  GT06  GT07  GT08  GT09  GT10  GT11  GT12  GT13  GT14  GT15  GT16  GT17
GL01  GL02  GL03  GL04  GL05  GL06  GL07  GL08  GL09  GL10  GL11  GL12  GL13  GL14  GL15  GL16  GL17

                                    'BEGIN' 'IDEN' A,B,C,A1,B1,C1 ::

  PZE     A

  PZE     B

  PZE     C

  PZE     A1

  PZE     B1

  PZE     C1

                                          A1=B1+C1 ::

  BGN

  ADD     B1      C1      GT01

  STO     GT01    A1

                                      'IF' A 'THEN' B=C 'ELSE'

  TRF     A       GL01

  STO     C       B

                                            'IF' B 'THEN' 'GOTO' SORTIE 'ELSE' C=B ::

  TRA     GL02

  LAB     GL01

  TRF     B       GL03
```

../..

```
TRA     GL04
LAB     GL03
STO     B       C
LAB     GL04
LAB     GL02

LAB     SORTIE
EXP     B1      C1      GT01
MPY     A1      GT01    GT02
STO     GT02    B1

DIV     B1      A1      GT01
STO     GT01    C1
SUB     B1      C1      GT01
STO     GT01    A1

ADD     A1      C1      GT01
STO     GT01    A1
END
```

```
SORTIE : B1=A1*B1**C1 ::

'BEGIN' C1=B1/A1 :: A1=B1-C1 'END' ::

A1=A1+C1
'END' ::
```

NVAR	NCPRIDLIST	NRS	NR1	NGT	NGL
1	1	7	1	1	3	6

16 HEURES 55 MINUTES 41 SECONDES

On cherche ensuite à éliminer certaines redondances dans les règles d'une p s g. Pour cela, on introduit la notion de p s g "admissible" ; on a pu aussi démontrer que toute p s g peut se ramener à ce type de grammaire.

Définition 2. Une p s g est "admissible" si elle est nulle ($V_T = \mathcal{R} = \emptyset$, $V_N = \{S\}$) ou si:

- pour tout $A \in V_T \cup V_N$, $A \neq S$, il existe $\alpha, \beta \in cl(V_T \cup V_N)$ tels que $S \xrightarrow{*} \alpha A \beta$
- pour tout Z, il existe une chaîne $\gamma \in V_T \; cl(V_T)$ telle que

$$Z \xrightarrow{*} \gamma$$

Une telle structure exclut :

- les symboles qu'on ne peut atteindre de S, ou ne figurant pas dans une chaîne terminale.
- qu'un symbole non-terminal ne puisse conduire à une chaîne terminale.

Pour qu'une p s g soit sous forme standard, il faut et il suffit que :

- $A_1 \in V_T$ (cf.Def.1) (1)
- $A_i \in V_N$ $i = 2,\ldots,n$ (2)

La condition (2) peut être facilement satisfaite. Si B est un nouveau symbole non terminal, toute règle de la forme :

$Z \longrightarrow a \; \alpha \; b \; \beta$ α, β étant des chaînes sur $V_T \cup V_N \cup \{\Delta\}$

peut être remplacée par :

$$Z \longrightarrow a\ \alpha\ B\ \beta$$
$$B \longrightarrow b$$

La première condition est plus difficile à remplir.

Pour faciliter la poursuite de l'exposé, donnons quelques définitions supplémentaires.

- A est dit tête de la règle générale :

$$Z \longrightarrow A\ Y_1 \ldots Y_n \quad , A, Y_1, \ldots, Y_n \in V_N \cup V_T$$

- Les règles dont Z est la partie gauche sont dites règles de Z.

- Pour une p s g G1

$$H(G1) = \{A \mid A \text{ est la tête d'une règle}\}$$
$$M(G1) = \{Z \mid \text{la tête de } Z \text{ appartient à } V_N\}$$

Ainsi, une p s g G1 "admissible" sous forme standard est telle que :

$$H(G1) = V_T$$
$$M(G1) = \emptyset$$

Les deux lemmes suivants montrent qu'il est possible de mettre sous la forme standard les règles d'un élément non terminal sans introduire pour cela de nouvelles ambiguïtés. (Cf. les articles de S. GREIBACH [Ref.3]).

Lemme 1. Etant donné une p s g "admissible" $G1 = (V_N, V_T, S, \mathfrak{R})$ et $Z \in V_N$, on peut toujours trouver une expression régulière R sur $V_T \cup V_N$ telle que :

- si R désigne α, $Z \xrightarrow{*} \alpha$
- si $Z \xrightarrow{*} \alpha$, α étant une chaîne terminale :
  - . ou R désigne α
  - . ou il existe une chaîne β désignée par R, telle que

$$Z \xrightarrow{*} \beta \xrightarrow{*} \alpha$$

- R est de la forme :

$$(a_1 \alpha_1 + \ldots + a_m \alpha_m + b_1 \beta_1 + \ldots + b_r \beta_r)\ cl(Q)$$

avec $a_i, b_i \in V_T$, α_i chaîne sur $V_T \cup V_N$ (peut être nulle), Q, β_i expressions régulières sur $V_T \cup V_N$ (Q peut être = Δ), $m \geqslant 0$, $r \geqslant 0$, $m+r > 0$.

Lemme 2. Etant donné une p s g admissible $G1 = (V_N, V_T, S, \mathfrak{R})$ et $Z \in V_N$, $Z \in M(G1)$, il existe une p s g admissible $G'1=(V'_N, V_T, S, \mathfrak{R}')$ tel que :

- $L(G'1) = L(G1)$
- $Z \notin H(G'1)$ et $Z \notin M(G'1)$ c.à.d. Z ne peut être une tête et les règles de Z ne possèdent que des têtes terminales.
- $V'_N \cap H(G'1) \subset V_N$ c.à.d. aucun nouveau symbole non terminal n'apparaît comme tête.
- $V'_N \cap H(G'1) \subseteq M(G'1)$ c.à.d. une tête est terminale ou sujet d'une règle non sous forme standard.
- $M(G'1) \cap V_N \subsetneq M(G1)$ c.à.d. les éléments de V_N sujets de règles non sous forme standard sont moins nombreux.

Nous allons monter sur un exemple comment transformer

une p s g admissible à l'aide de ces deux lemmes.

Ex: Soit G1 une p s g admissible telle que :

$V_T = \{a, b, d, e\}$ $\quad V_N = \{S, Y\}$

$S \longrightarrow a$

$S \longrightarrow S\ b$

$S \longrightarrow Y\ a$

$Y \longrightarrow e$

$Y \longrightarrow Y\ d\ Y$

A partir de S cherchons l'expression régulière du lemme 1. Trois cas se présentent :

1. $S \longrightarrow a$
2. Récursivité sur S

$S \longrightarrow S\ b$

Nous pouvons écrire : $S \xrightarrow{*} S\ cl(b)$ (cl(b)non vide)

3. Récursivité sur Y

$S \longrightarrow Y\ a$

Nous devons tenir compte des règles :

$Y \longrightarrow Y\ d\ Y$ qui donnent $Y \xrightarrow{*} Y\ cl(d\ Y)$

$Y \longrightarrow e$ $\quad Y \longrightarrow e$ (cl(dY)non vide)

d'où : $S \longrightarrow e\ cl(d\ Y)a$

Finalement, on peut atteindre de S une chaîne désignée par l'expression régulière :

$$(a + e\ cl(dY)a)\ cl(b)$$

A partir de cette expression régulière, construisons la nouvelle p s g admissible G'1 ne contenant pas de règles de S non sous forme standard. cl(d Y) et cl(b) (non-vides) sont générés respectivement par les nouveaux symboles non-terminaux B et D suivant les règles :

$B \longrightarrow b$ $\qquad$ $D \longrightarrow d\ Y$

$B \longrightarrow b\ B$ $\qquad$ $D \longrightarrow d\ Y\ D$

et les règles de S s'écrivent :

$S \longrightarrow a$ $\qquad$ $S \longrightarrow e\ D\ a$ $\qquad$ $S \longrightarrow e\ a$

$S \longrightarrow a\ B$ $\qquad$ $S \longrightarrow e\ D\ a\ B$ $\qquad$ $S \longrightarrow e\ a\ B$

On remarque ainsi qu'en créant de nouvelles règles et de nouveaux symboles non-terminaux convenables, on peut éviter les règles récursives à gauche.

A l'aide de ces deux lemmes, S. GREIBACH [Ref.13] démontre l'équivalence d'une grammaire context-free et d'une grammaire sous forme standard. Intuitivement, le premier lemme montre qu'il est possible d'éliminer les règles non sous forme standard correspondant à un élément non terminal, le second qu'en appliquant successivement cette transformation, le processus converge.

Ainsi, la p s g sous forme standard équivalente à celle donnée en exemple est $G^* = \{V_N, V_T, S, \mathfrak{R}\}$ telle que

$V_N = \{S, Y, A, B, D\}$ $\qquad$ $V_T = \{a, b, d, e\}$

$S \longrightarrow a$ $\qquad$ $B \longrightarrow b$ $\qquad$ $Y \longrightarrow e$

$S \longrightarrow e\ D\ A$ $\qquad$ $B \longrightarrow b\ B$ $\qquad$ $Y \longrightarrow e\ D$

$S \longrightarrow a\ B$ $\qquad$ $D \longrightarrow d\ Y$ $\qquad$ $A \longrightarrow a$

$S \longrightarrow e\ D\ A\ B$ $\qquad$ $D \longrightarrow d\ Y\ D$

$S \longrightarrow e\ A$

$S \longrightarrow e\ A\ B$

III.2. APERCU GENERAL SUR L'ALGORITHME D'ANALYSE PREDICTIVE UTILISANT UN REORDONNANCEMENT DES REGLES.

Décrivons maintenant l'algorithme d'analyse inspiré de Kuno-Oettinger [Ref.11] que nous avons expérimenté (nous verrons ensuite comment y adopter un processus ordonnant les règles de grammaire).

Les ambiguïtés syntaxiques sont ordinaires dans les langages naturels : le but de Kuno-Oettinger était de les détecter en déterminant toutes les analyses associées à une chaîne donnée. De par leur construction, les langages de programmation sont non-ambigus, mais seule l'utilisation effective d'un langage context-free peut confirmer l'inexistence d'ambiguïtés. La plupart des analyseurs employés dans les compilateurs dirigés par la syntaxe (syntax directed) présupposent l'absence d'ambiguïtés et considèrent seulement le résultat fourni par la première analyse. L'algorithme de Kuno-Oettinger peut être aisément modifié pour s'arrêter au résultat de cette première analyse et c'est un tel analyseur que nous utilisons effectivement. Les raisons qui nous ont fait étudier cet algorithme sont principalement :

1. Cet analyseur est étroitement lié à la classification sélective de Griffiths et Petrick [Ref.12] et analyse plus rapidement que les algorithmes descendants habituels.
2. Il est relativement simple surtout parce qu'il exige que les règles de grammaire soient données sous forme standard.

Cela rend particulièrement facile sa présentation.

Il est assez pratique d'exprimer les règles de ce genre sous la forme :

$$(Z\ ,\ a) \longrightarrow Y_1\ Y_2 \ldots Y_m \qquad (1a)$$
$$\text{et } (Z\ ,\ a) \longrightarrow \Delta \qquad (2a)$$

L'algorithme d'analyse utilise des piles (ou PDS de push down store) et peut être brièvement décrit comme suit :

a. Considérons le couple formé par la métavariable "Z" située au sommet de la pile et soit "a" le symbole de base considéré dans la chaîne d'entrée. Trois cas sont alors possibles :

1. Il existe seulement une règle de la forme :

$$(Z\ ,\ a) \longrightarrow Y_1\ Y_2 \ldots, Y_m$$

dans la grammaire considérée. Dans ce cas, Z est remplacé par Y_m ; Y_{m-1} ,...., Y_2, Y_1 sont placés dans la pile, Y_1 étant au sommet. On répète alors le processus décrit en a. Quand la règle (X, a) est du type 2a, l'opération effectuée est équivalente à une élimination de Z du sommet de la pile. Dans les deux cas, on augmente de 1 la valeur du pointeur de la chaîne d'entrée (cette chaîne est examinée de gauche à droite).

2. Il existe plusieurs règles (X, a). On crée alors une nouvelle pile pour chaque règle en recopiant dans chacune le contenu de la pile originale. Et on répète pour

ces piles le processus a.

3. Il n'existe pas de règle (X, a) : on abandonne tout traitement sur la pile considérée. L'algorithme prend en charge la pile suivante qui pourrait avoir été formée en 2.

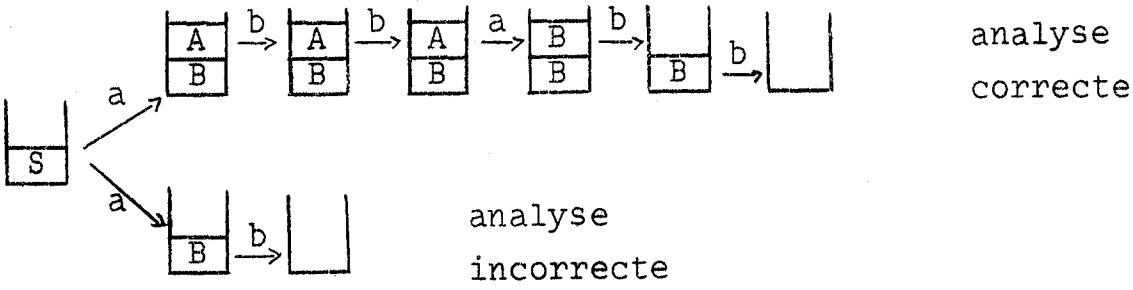
b. Une analyse unique est achevée lorsqu'après une suite d'itérations décrites en a., on a examiné tous les éléments de la chaîne d'entrée et que la pile courante est vide (si on désire trouver toutes les analyses possibles, la condition supplémentaire est qu'il ne reste plus de pile à considérer).

Exemple : Soit la grammaire $G2^{*} = \{V_N, V_T, S, \mathfrak{R}\}$ écrite sous la forme standard :

$V_T = \{a, b\}$ et $V_N = \{S, A, B\}$

(S, a) —> AB	(A, b) —> A	(B, a) —> B
(S, a) —> B	(A, a) —> B	(B, b) —> Δ

et soit à analyser la chaîne a b b a b b. La figure suivante indique le contenu de deux piles durant l'analyse.



Notons que la description ci-dessus est pédagogique

et que deux piles au plus sont nécessaires pour une construction effective de l'algorithme. C'est ce que nous montrons au chapitre IV.2.

Dans le cas d'une seule analyse, on se rend compte que l'ordre dans lequel on consulte les règles de grammaire joue un grand rôle quant à l'efficacité de l'analyse ; en effet, comme on opère par essais successifs, le temps d'analyse est évidemment réduit si les règles les plus courantes sont les premières à être essayées.

IRONS [Ref.17] remarque que le problème d'ordonnancement optimal des règles n'est pas trivial et suggère même qu'il soit laissé "à la méditation pendant les jours de pluie"... A notre connaissance, il n'existe pas de solution théorique à ce problème qui soit effectivement utilisable. Notre solution, essentiellement pratique, consiste à développer un programme qui enregistre d'abord le nombre d'utilisations heureuses d'une règle. Dans l'étape suivante, et à la volonté de l'utilisateur, le programme réordonne l'enchaînement des règles afin que les premières considérées dans une analyse ultérieure soient les plus fréquemment utilisées.

On comprend que l'ordonnancement des règles est particulièrement important pour l'algorithme d'analyse précédemment décrit ; en effet, le nombre de règles à consulter est toujours plus élevé que pour l'algorithme de Cheatham (Chap.II). De plus, un tel processus de réordonnancement y est facilement incorporable ; il pourrait cependant l'être aussi à d'autre types d'analyseurs. Le programme ALGOL réalisant ce réordonnancement est présenté au chapitre IV.3.

CHAPITRE IV

REALISATION D'UN ANALYSEUR POUR UN LANGAGE DU GENRE ALGOL

Nous présentons dans ce chapitre les programmes ALGOL décrivant les algorithmes cités dans le chapitre III, c.à.d. :

- L'algorithme transformant une grammaire "context-free" en une grammaire équivalente sous forme standard. A vrai dire, comme nous le détaillons plus loin, afin d'éviter un trop grand nombre de règles, on tolère l'existence de règles particulières (non sous forme standard) dans cette grammaire transformée.

- L'algorithme d'analyse descendante unique inspiré de Kuno-Oettinger.

- L'algorithme permettant un réordonnancement des règles en fonction d'une statistique sur leurs fréquences respectives d'utilisation.

Dans la dernière partie, on compare les temps d'analyse obtenus pour diverses chaînes.

IV.1. TRANSFORMATION D'UNE GRAMMAIRE CONTEXT-FREE EN UNE GRAMMAIRE EQUIVALENTE SOUS UNE FORME PROCHE DE LA FORME STANDARD.

Signalons tout d'abord que cet algorithme ne s'applique qu'à des grammaires ne possédant pas de règles récursives à gauche : nous savons que ceci n'implique aucune restriction sur la grammaire considérée. Il n'est alors pas nécessaire de passer par l'intermédiaire de l'expression régulière du chapitre III (p.III.6).

C'est à l'aide de l'exemple suivant que nous allons montrer le fonctionnement de l'algorithme.

Exemple : Soit la grammaire G3 = $\{V_N, V_T, S, \mathcal{R}\}$

$V_T = \{\not\#, a, b, c, d, e, f\}$

$V_N = \{S, A, B, C, D, E, F\}$

$S \longrightarrow \not\# A \not\#$

$A \longrightarrow a\ E \mid B$

$B \longrightarrow C \mid C\ b\ B$

$C \longrightarrow c \mid D$

$D \longrightarrow d \mid d\ D$

$E \longrightarrow F \mid F\ e\ E$

$F \longrightarrow C \mid A\ f\ A$

Le symbole "|" a la signification de la forme normale de Backus.

Il suffit de remplacer tout élément non terminal trouvé en tête d'une partie droite par les parties droites des règles de cet élément. Et ceci jusqu'à ce que les nouvel-

les règles ainsi produites soient sous forme standard. A vrai dire, par un abus de langage, nous appellerons tout simplement règles sous forme standard ou standard toute règle dont la tête est un élément terminal. Nous nous en justifierons du point de vue de l'analyse au § 2 de ce chapitre.

Exemple : B —> c b B sera dite standard.

Ainsi, les règles de G3

B —> C | C b B se transforment successivement en :

B —> c | D | c b B | D b B

B —> c | d | d D | c b B | d b B | d D b B

qui sont sous forme standard. Nous remarquons que les non terminaux tels que C qui ont été remplacés n'apparaissent plus dans la grammaire transformée : il est inutile donc de traiter les règles dont ils sont sujets. Toutefois cette disparition peut poser des problèmes dans une phase de génération (cf.Conclusion).

Dans cet exemple, le nombre de règles a triplé. Dans le cas de la règle :

E —> F | F e E

il serait multiplié par dix. On comprend que la grammaire transformée d'une grammaire genre ALGOL contiendrait un nombre considérable de règles, en conséquence nous avons décidé de laisser à l'utilisateur le choix de garder certaines règles, appelées règles particulières, non transformées. Supposons

que nous désirons conserver la règle :

$$A \longrightarrow B$$

Dans la grammaire donnée, on remplacera alors B par un symbole terminal fictif x. Ceci permet de ne pas modifier la partie droite de cette règle et dans la grammaire transformée, x reprendra sa signification première B.

Un choix possible sur G3 donnerait la grammaire G3* transformée suivante :

Règles standard	Règles particulières
(S,#) —> A #	
(A, a) —> E	A —> B
(B, c) —> Δ | bB	
(B, d) —> Δ | D|bB|DbB	
(D, d) —> Δ | D	
(E, c) —> Δ | eE	
(E, d) —> Δ | D|eE|DeE	E —> AfA|AfAeE

Nous appelerons partie droite d'une règle standard la chaîne située à droite du signe —> lorsque la règle est écrite sous la forme ci-dessus.

Voyons maintenant quelle est la représentation en machine de la grammaire donnée et des informations s'y rapportant dont nous avons besoin. On utilise les six tableaux à une dimension suivants :

RACINE d'indice ELEM NON TERM
identique à celui utilisé précédemment (cf.chap.II.1)

ARBRE d'indice K

analogue à celui utilisé précédemment (Cf.Chap.II.1) mais comportant au début de chaque règle le nombre d'éléments en partie droite et au début de chaque série de règles relatives au même sujet le nombre de règles de la série.
De plus, les éléments de règles sont disposés dans l'ordre inverse de l'ordre de lecture afin de faciliter les opérations sur les piles (Cf.Chap.IV.2).

TABET d'indice IT

contenant les têtes de règles constituées par des éléments terminaux dans l'ordre de lecture des règles (non compris les éléments terminaux fictifs).

TABENT d'indice INT

contenant les éléments non terminaux situés au moins en deuxième position d'une partie droite c.à.d. sujets de règles dans la grammaire transformée. Ce tableau est utilisé dans la partie de réordonnancement.

TABET 1 et TABENT1 d'indices IT1 et INT1

contenant respectivement les éléments terminaux fictifs et les éléments non-terminaux correspondants.

Ce rangement est effectué évidemment par programme. Les règles, sous forme de Backus avec leurs éléments préalablement codés, sont lues en séquence.

La représentation de la grammaire transformée et des

informations nécessaires au fonctionnement des deux algorithmes suivants nécessité neuf tableaux :

REGLE à deux dimensions d'indice ENT (élément non-terminal) et ET (élément terminal). Si le couple (ENT, ET) conduit à une prédiction, REGLE [ENT,ET] fournit l'adresse J dans le tableau TABLE du début des règles standard se rapportant à ce couple.

TABLE d'indice J

contient les parties droites des règles standard et particulières disposées dans l'ordre inverse comme dans ARBRE. Au début de chaque série de règles admettant un même couple (élément non terminal, l'élément terminal) ou un même sujet, on indique :

- le nombre de règles.
- l'adresse J de la première règle à consulter.

On précise aussi au début de chaque règle le nombre d'éléments de sa partie droite et à la fin l'adresse J de la règle suivante de la même "série" (ces adresses sont appelées données de liaison). S'il n'y a pas de règles suivantes, on trouve ETOILE au lieu de cette adresse.

Exemple : pour la série (E, c) $\longrightarrow \Delta$

(E, c) $\longrightarrow$ e E

de la grammaire $G3^{*}$, les données de liaison dans le tableau de la page IV.12 sont respectivement 19 et 37.

LMINIMUM et LMAXIMUM d'indice ENT (élément non terminal) fournissent respectivement les adresses L minimum et maximum de TABADRESSE entre lesquelles se trouvent toutes les adresses de début de parties droites se rapportant à ENT.

ELEMTERM et TABADRESSE d'indice L
contenant respectivement les symboles terminaux formant avec un symbole non terminal la partie gauche d'une règle standard, et l'adresse J dans TABLE du début de cette règle.

RAC PART d'indice ENT
indique où débute dans TABLE la règle particulière dont ENT est le sujet.

LMINIPART, LMAXIPART d'indice ENT et TABADRPART
d'indice LP jouent un rôle analogue pour les règles particulières à celui de LMINIMUM et LMAXIMUM et TAB ADRESSE pour les règles standard.

Les pages suivantes fournissent les représentations de la grammaire G3 et de sa grammaire transformée $G3^{*}$. L'algorithme de transformation est donné ensuite en ALGOL ;il ne comprend pas la phase préliminaire de construction des tableaux représentant la grammaire G3.

On utilise deux piles STACK et STACK STACK dont les pointeurs IS et ISS repèrent les sommets. STACK contient la

partie droite de la règle à transformer avec l'élément non terminal à remplacer en tête. Pour une série de règles, STACK STACK retient les informations nécessaires à un traitement ultérieur des règles suivantes. Enfin, le tableau à deux entrées REGSUIV indique pour un couple (élément non terminal, élément terminal) l'adresse dans TABLE où doit être placée une éventuelle donnée de liaison dans le cas du traitement d'une règle appartenant à une série ; le tableau RAC PART SUIV joue le même rôle pour les règles particulières.

REPRESENTATION DE LA GRAMMAIRE G3

	K	ARBRE	Commentaires
S	①	1	nombre de parties droites de S
	2	3	nombre d'éléments
	3	#	
	4	A	S ⟶ # A #
	5	#	
A	⑥	2	nbre de partie drte
	7	2	nombre d'éléments
	8	E	
	9	a	A ⟶ a E
	10	1	nombre d'éléments
	11	y	A ⟶ y
B	⑫	2	n. p. d.
	13	1	n. e.
	14	C	B ⟶ C
	15	3	n. e.
	16	B	
	17	b	B ⟶ C b B
	18	C	
C	⑲	2	n. p. d.
	20	1	n. e.
	21	c	C ⟶ c
	22	1	n. e.
	23	D	C ⟶ D
D	㉔	2	n. p. d.
	25	1	n. e.
	26	d	D ⟶ d
	27	2	n. e.
	28	D	
	29	d	D ⟶ d D
E	㉚	2	n. p. d.
	31	1	n. e.
	32	F	E ⟶ F
	33	3	n. e.
	34	E	
	35	e	E ⟶ F e E
	36	F	
F	㊲	2	n. p. d.
	38	1	n. e.
	39	C	F ⟶ C
	40	3	n. e.
	41	A	
	42	f	F ⟶ x f A
	43	x	

ELEMNONTERM	RACINE
S	1
A	6
B	12
C	19
D	24
E	30
F	37

IT	TABET
1	#
2	a
3	c
4	d

INT	TABENT
1	S
2	A
3	E
4	B
5	D

IT1	TABET1	TABENT1
1	x	A
2	y	B

REPRESENTATION DE LA GRAMMAIRE G3*

REGLE [ENT, ET]

ET / ENT	#	a	b	c	d	e	f
S	1	*	*	*	*	*	*
A	*	7	*	*	*	*	*
B	*	*	*	57	62	*	*
C	*	*	*	*	*	*	*
D	*	*	*	*	83	*	*
E	*	*	*	17	22	*	*
F	*	*	*	*	*	*	*

ENT	LMINIMUM	LMAXIMUM
S	1	1
A	2	2
B	9	14
C	*	*
D	15	16
E	3	8
F	*	*

	L	ELEMTERM	TABADRESSE	Commentaires
S	1	#	3	(S,#)⟶A #
A	2	a	9	(A,a)⟶E
E	3	c	19	(E,c)⟶Δ
	4	d	24	(E,d)⟶Δ
	5	d	27	(E,d)⟶D
	6	c	37	(E,c)⟶eE
	7	d	41	(E,d)⟶eE
	8	d	45	(E,d)⟶DeE
B	9	e	59	(B,c)⟶Δ
	10	d	64	(B,d)⟶Δ
	11	d	67	(B,d)⟶D
	12	c	70	(B,c)⟶bB
	13	d	74	(B,d)⟶bB
	14	d	78	(B,d)⟶DbB
D	15	d	85	(D,d)⟶Δ
	16	d	88	(D,d)⟶D

ENT	RACPART	LMINIPART	LMAXIPART
A	12	1	1
E	30	2	3

	LP	TABABRPART	Commentaires
A	1	14	A ⟶ B
E	2	32	E ⟶ AfA
	3	50	E ⟶ AfAeE

TABLE CONTENANT TOUTES LES REGLES NORMALES ET PARTICULIERES

	J	TABLE	Commentaires
(S,#)→	(1)	1	NR (nbre de règles)
	2	3	données de liaison (d.l.)
	3	2	NE (nbre d'éléments)
	4	#	(S,#) ⟶ A #
	5	A	
	6	*	Fin des règles (S,#)
(A,a)→	(7)	1	NR
	8	9	d.l.
	9	1	NE
	10	E	(A,a) ⟶ E
	11	*	Fin des regles de (A,a)
A ⟶ (règles particulieres)	12	1	NR
	13	14	d.l.
	14	1	NE
	15	B	A ⟶ B
	16	*	Fin des règles particulières de A
(E,c)→	(17)	2	NR
	18	19	d.l.
	19	1	NE
	20	Δ	(E,c) ⟶ Δ
	21	37	d.l.
(E,d)→	(22)	4	NR
	23	24	d.l.
	24	1	NE
	25	Δ	(E,d) ⟶ Δ
	26	27	d.l.
	27	1	NE
	28	D	(E,d) ⟶ D
	29	41	
E ⟶ (règle particulière)	30	2	NR

	J	TABLE	Commentaires
	31	32	d.l.
	32	3	NE
	33	A	
	34	f	E ⟶ AfA (règle particulière)
	35	A	
	36	50	d.l.
	37	2	NE
	38	E	(E,c) ⟶ eE
	39	e	
	40	*	Fin des règles (E,c)
	41	2	NE
	42	E	(E,d) ⟶ eE
	43	e	
	44	45	d.l.
	45	3	NE
	46	E	
	47	e	(E,d) ⟶ DeE
	48	D	
	49	*	Fin des règles (E,d)
	50	5	NE
	51	E	
	52	e	
	53	A	E ⟶ AfAeE
	54	f	
	55	A	
	56	*	Fin des règles particulières de E
(B,c)→	(57)	2	NR
	58	59	d.l.
	59	1	NE
	60	Δ	(B,c) ⟶ Δ

	J	TABLE	Commentaires
	61	70	d.l.
(B,d)→	(62)	4	NR
	63	64	d.l.
	64	1	NE
	65	Δ	(B,d) ⟶ Δ
	66	67	d.l.
	67	1	NE
	68	D	(B,d) ⟶ D
	69	74	d.l.
	70	2	NE
	71	B	(B,c) ⟶ bB
	72	b	
	73	*	Fin des règles de (B,c)
	74	2	NE
	75	B	(B,d) ⟶ bB
	76	b	
	77	78	
	78	3	NE
	79	B	(B,d) ⟶ DbB
	80	b	
	81	D	
	82	*	Fin des règles de (B,d)
(D,d)→	(83)	2	NR
	84	85	d.l.
	85	1	NE
	86	Δ	(D,d) ⟶ Δ
	87	88	d.l.
	88	1	NE
	89	D	(D,d) ⟶ D
	90	*	Fin des règles de (D,d)

ALGORITHME DE TRANSFORMATION DE GRAMMAIRE

```
PREP FORM STANDARD :

J:=L:=LP:=ISS:=0 ;
pour INT:=1 pas 1 jusqua INTM faire
debut IS:=0 ; H:=ENT:=TABENT[INT] ;
    LIMINIMUM[ENT]:=L+1 ; LIMINIPART[ENT]:=LP+1 ;
 TRANSFORM NONTERM :
    K:=RACINE[H] ; NR:=ARBRE[K] ;
    K:=K+1 ; NE:=ARBRE[K] ;
    si  NR > 1 alors
    pour V:=IS, K+NE+1, NR-1 faire
    debut ISS:=ISS+1 ; STACK STACK[ISS]:=V
    fin d'empiler dans STACK STACK les informations néces-
    saires pour former les règles suivantes ;
 PREP EMPSTACK :
    COPIE(ARBRE[K+1], STACK[IS+1], NE) ; IS:=IS+NE ;
    H:=STACK[IS] ; IS:=IS-1 ;
    si ¬ TERMINAL[H] alors allera TRANSFORM NONTERM ;
 PREP REGLE :
    si  TERMINAL FICTIF[H] alors allera PREP REG PARTICULIERE ;
 PREP REG NORMALE :
    L:=L+1 ; ELEMTERM[L]:=H ;
    si REGLE[ENT,H]= ETOILE alors
    debut J:=J+1 ; REGLE[ENT,H]:=J ; TABLE[J]:=1 ;
        J:=J+1 ; TABLE[J]:=TABADRESSE[L]:=J+1
    fin sinon
    debut TABLE[REGLE[ENT,H]]:=TABLE[REGLE[ENT,H]]+1 ;
        TABLE[REG SUIV[ENT,H]]:=TABADRESSE[L]:=J+1
    fin ;
```

```
    si IS = 0 alors
    debut J:=J+1 ; TABLE[J]:=1 ;
          J:=J+1 ; TABLE[J]:=DELTA
    fin sinon
    debut J:=J+1 ; TABLE[J]:=IS ;
          COPIE (STACK[1], TABLE[J+1],IS) ; J:=J+IS
    fin ;
    J:=J+1 ; REG SUIV[ENT,H]:=J ;
    allera PREP REG SUIVANTE ;
PREP REG PARTICULIERE :
    pour V:=1 pas 1 jusqua ITM1 faire
    si H=TABET1[V] alors
    debut LP:=LP+1 ; IS:=IS+1 ; STACK[IS]:=TABENT1[V] ;
          si RAC PART[ENT] = ETOILE alors
          debut J:=J+1 ; RAC PART[ENT]:=J ; TABLE[J]:=1 ;
                J:=J+1 ; TABLE[J]:=TABADR PART[LP]:=J+1
          fin sinon
          debut TABLE[RAC PART[ENT]]:=TABLE[RAC PART[ENT]+1 ;
                TABLE[RAC PART SUIV[ENT]]:=TABADR PART[LP]:=J+1
          fin ;
          J:=J+1 ; TABLE[J]:=IS ;
          COPIE (STACK[1], TABLE[J+1],IS) ; J:=J+IS ;
          J:=J+1 ; RAC PART SUIV[ENT]:=J
    fin de formation des règles particulières ;
PREP REG SUIVANTE :
    si ISS = 0 alors allera PREP FINREGLE ;
    NR:=STACK STACK[ISS] ;
```

```
    si NR = 0 alors
    debut ISS:=ISS-3 ; allera PREP REG SUIVANTE
    fin d'éliminer les informations concernant un élément
    non terminal ;
    STACK STACK [ISS] :=NR-1 ;
    K:=STACK STACK [ISS-1] ; NE:=ARBRE [K] ;
    STACK STACK [ISS-1] :=K+NE+1 ;
    IS:=STACK STACK [ISS-2] ;
    allera PREP EMP STACK ;
PREP FIN REGLE :
    si LMINIMUM [ENT] ⩽ L alors LMAXIMUM [ENT] :=L
                          sinon LMINIMUM [ENT] :=ETOILE ;
    si LMINIPART [ENT] ⩽ LP alors LMAXIPART [ENT] :=LP
                          sinon  LMINIPART [ENT] :=ETOILE
fin boucle pour INT et de formation des règles normales et
    particulières ;
```

N.B: COPIE est une procédure en code admettant trois paramètres A, B, N. Son but est de ranger les N premiers éléments du tableau A dans les N premières positions du tableau B. Elle emploie l'ordre TRANSMIT (du langage d'assemblage MAP) qui produit un recopiage d'une zône de mémoires dans une autre.

TERMINAL et TERMINAL FICTIF sont des tableaux booléens ; un élément de ces tableaux est vrai s'il correspond à un élément de vocabulaire satisfaisant au type indiqué.

INTM et ITM1 contiennent respectivement les valeurs maximales de INT et de IT1.

IV.2. DESCRIPTION DE L'ALGORITHME D'ANALYSE.

Il correspond à la présentation du chapitre III.2 si la grammaire est exactement sous forme standard. Lorsqu'un élément non terminal situé au sommet de la pile est sujet d'une règle particulière, on essaye d'abord d'appliquer les règles standard dont il est le sujet (s'il en existe). Si cet essai est infructueux, on place dans la pile la partie droite de cette règle particulière : dans le cas de plusieurs règles particulières, on crée une nouvelle pile pour chacune en y recopiant le contenu de la pile originale. Le pointeur de la chaîne d'entrée n'est, bien sur, pas avancé.

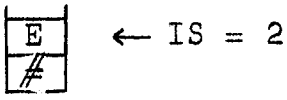
Lorsqu'un élément terminal se présente au sommet de la pile, on le compare avec l'élément de la chaîne d'entrée considérée : s'il lui est égal, l'analyse est correcte, on examine le symbole précédent dans la pile et le pointeur de la chaîne d'entrée avance de 1, sinon l'algorithme prend en charge la pile suivante. Cet élément terminal n'a pu être introduit que par l'application d'une règle non littéralement sous forme standard. Son apparition est très aisément et efficacement traitée : on justifie ainsi quelque peu l'abus de langage signalé au §.1 de ce chapitre.

Nous avons vu dans le paragraphe précédent comment la grammaire est représentée en machine. Les chaînes terminales à analyser sont stockées dans le tableau à une dimension CHAINE. Comme dans l'algorithme décrit au chapitre II.1. CHAINE[I] a pour valeur le code numérique correspondant à l'é-

lément du vocabulaire terminal situé à la ième position dans la chaîne, position repérée par le pointeur I.

En fait, on n'utilise que deux piles. La première est représentée par le tableau à une dimension STACK. Le pointeur IS en repère le sommet. Lorsqu'à priori plusieurs règles, qu'elles soient particulières ou non sont applicables, une pile auxiliaire appelée STACK STACK retient les informations propres à une remise à jour éventuelle de la pile STACK. ISS pointe sur le sommet de STACK STACK. Nous serons plus explicites en illustrant ce procédé par un exemple :

Considérons la chaîne # a c e d d d e c # satisfaisant à la grammaire G3* dont l'analyse est fournie à la page IV.24. A l'examen du troisième élément de la chaîne (c), la pile STACK est dans l'état suivant :



La pile STACK STACK à ce degré de l'analyse est encore vide (une seule règle possède S comme sujet et une seule règle standard admet le couple (A,a)).

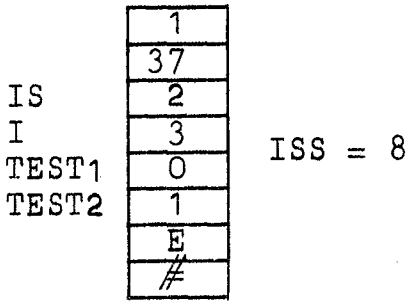
Les règles suivantes sont maintenant susceptibles d'être utilisées :

(E,c) —> Δ
(E,c) —> E
E —> A f A
E —> A f A e E

La priorité est donnée à l'essai des règles standard. Ce choix est sanctionné par les valeurs des indicateurs TEST1 et TEST2.

TEST1 = 0 signifie que nous essayons une règle standard (dans le cas contraire TEST1 = 1)
TEST2 = 1 signifie qu'il existe concurremment au moins une règle particulière (sinon TEST2 = 0)

Les informations stockées successivement dans la pile STACK STACK sont les suivantes : d'abord une copie de la pile STACK et ensuite TEST2, TEST1, les valeurs courantes des pointeurs I et IS, l'adresse dans TABLE du début de la règle suivant celle employée et de même type, enfin, le nombre de règles du même type susceptible d'être utilisées. Dans notre cas, STACK STACK a la structure suivante :



L'essai de la règle (E,c)$\longrightarrow\Delta$ se révélant plus tard infructueux, il faudra :

- replacer dans la pile STACK la partie précédemment recopiée dans STACK STACK.

- à l'aide des autres informations, choisir la règle à utiliser maintenant et réinitialiser le pointeur I.

Une simple remarque a permis d'introduire dans l'algorithme un test supplémentaire signalant une analyse incorrecte ; son efficacité aurait pu en être améliorée (Cf. Conclusion). En effet, lorsque le nombre d'éléments contenus dans la pile STACK est supérieur au nombre d'éléments de la chaîne d'entrée restant à examiner, il est inutile de poursuivre l'analyse.

Les résultats sont contenus dans les quatre tableaux à une dimension suivants :

NONTERM d'indice I
fournit, si on a appliqué une règle standard pour reconnaître le ième élément de la chaîne d'entrée, le sujet de cette règle.

ADRESSE d'indice I
contient l'adresse dans TABLE de la précédente règle.

NONTERMPART et ADRESSEPART d'indice I
fournissent respectivement les mêmes indications que NONTERM et ADRESSE, mais dans le cas de l'application d'une règle particulière.

On détient donc grâce à ces tableaux, la structure syntaxique de la chaîne donnée. Dans les pages suivantes, à la suite de l'algorithme, nous donnons l'arbre syntaxique correspondant à trois chaînes satisfaisant à la grammaire G3*.

ALGORITHME D'ANALYSE.

```
INITIALISATION :
    I:=ISS:=0 ; IS:=1 ; STACK[IS]:=AXIOME ;
    si REGLE[AXIOME, CHAINE[1]] = ETOILE alors allera ERREUR ;
CONT AVANT :
    si IS > IFINAL - I alors  allera ANALYSUIVANTE ;
    I:=I+1 ;
    si TERMINAL[STACK[IS]] alors
    debut
    |   si STACK[IS] ≠ CHAINE[I] alors allera ANALYSUIVANTE ;
    |   NONTERM[I]:=ETOILE ; allera REGRES STACK
    fin cas où la prédiction au sommet du stack est terminale ;
    J:=REGLE[STACK[IS], CHAINE[I]] ;
    si RACPART[STACK[IS]]=ETOILE alors
    debut
    |   si J = ETOILE alors  allera  ANALYSUIVANTE ;
    |   TEST1 := TEST2 := 0 ; allera PREDNONTERM
    fin cas normal ;
    si J = ETOILE alors
    debut I:=I-1 ; J:=RAC PART[STACK[IS]]; TEST1 := 1;
    |   si NONTERMPART[I+1]= ETOILE alors
    |   debut NONTERMPART[I+1]:=STACK[IS] ;
    |   |     ADRESSEPART[I+1]:= TABLE[J+1]
    |   fin sinon
    |   si NONTERMPART[I+1]= STACK[IS] alors
    |     ADRESSE PART[I+1]:=TABLE[J+1] ;
    |   allera PREDNONTERM
    fin cas particulier ;
```

```
    TEST1 := 0 ; TEST2 := 1 ;
    NR := TABLE[J]; J := TABLE[J+1] ; NE := TABLE[J] ;
    allera EMP STACK STACK ;
PRED NON TERM :
    NR := TABLE[J]; J:=TABLE[J+1] ; NE := TABLE[J] ;
    si NR> 1 alors
EMP STACK STACK :
    debut COPIE (STACK[1], STACK STACK[ISS+1], IS) ;
        |   ISS := ISS+IS ;
        |   pour V:=TEST2, TEST1, I, IS, TABLE[J+NE+1], NR-1 faire
        |   debut ISS:=ISS+1 ; STACK STACK[ISS]:=V
        |   fin
    fin d'empiler dans le stack du stack les informations con-
        cernant la règle suivante ;
    si TEST1 ≠ 0 alors allera EMPILER STACK ;
CONT EMPILER :
    NONTERM[I]:= STACK[IS] ; ADRESSE[I]:= J ;
    si TABLE[J+1]= DELTA alors allera REGRES STACK ;
EMPILER STACK :
    IS := IS-1 ;
    COPIE (TABLE[J+1], STACK[IS+1], NE) ;
    IS := IS+NE ; allera CONTAVANT ;
ANALYSUIVANTE :
    si ISS = 0 alors allera ERREUR ;
    NR := STACK STACK[ISS]; IS:=STACK STACK[ISS-2]; IM:=I ;
```

```
si NR = 0 alors
debut U := ISS ; ISS := ISS-IS-6 ;
    si STACK STACK [U-4]≠ 0 ∨ STACK STACK [U-5]= 0 alors
                                allera ANALYSUIVANTE ;
    COPIE (STACK STACK [ISS+1], STACK [1], IS) ;
    I:=STACK STACK [U-3]-1 ; J:=RACPART [STACK [IS]] ;
    NONTERMPART [I+1] :=STACK [IS] ; ADRESSEPART [I+1]:=TABLE [J+1] ;
    pour V:=I+2 pas 1 jusqua IM faire
    NONTERMPART [V] :=ADRESSEPART [V] :=ETOILE ;
    TEST1 :=1 ; allera PREDNONTERM
fin de désempiler le stack du stack ;
STACK STACK [ISS]:= NR-1 ; U:=ISS-IS-6 ;
COPIE (STACK STACK [U+1], STACK [1], IS) ;
I:=STACK STACK [ISS-3]; J:=STACK STACK [ISS-1]; NE:=TABLE [J];
STACK STACK [ISS-1] :=TABLE [J+NE+1] ;
si STACK STACK [ISS-4]= 0 alors allera CONTEMPILER ;
NONTERMPART [I+1] :=STACK [IS]; ADRESSEPART [I+1]:=J ;
pour V:=I+2 pas 1 jusqua IM faire
NONTERMPART [V]:=ADRESSEPART [V]:=ETOILE ;
allera EMPILER STACK ;
REGRES STACK :
IS := IS-1 ;
si IS > 0 alors allera CONTAVANT ;
si I < IFINAL alors allera  ANALYSUIVANTE ;
```

N.B. L'étiquette ERREUR renvoie à une partie de programme qui signale l'erreur avant de s'arrêter.

EXEMPLES ET RESULTATS

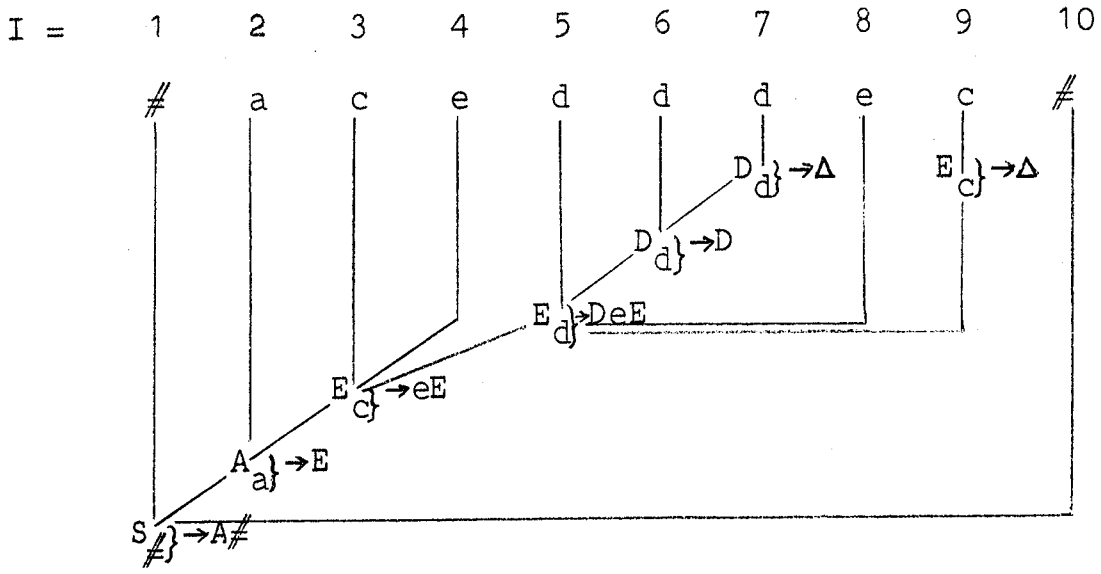
Exemple I : Utilisation des règles standards.

I	1	2	3	4	5	6	7	8	9	10
CHAINE	#	a	c	e	d	d	d	e	c	#

Résultats d'analyse (règles standards)

I	NONTERM	TERMINAL	PREDICTIONS	Commentaires
1	S	#	A #	(S,#)—>A #
2	A	a	E	(A,a)—>E
3	E	c	e E	(E,c)—>e E
4	*	e		prédiction terminale e identique à l'élément e de la chaîne
5	E	d	D e E	(E,d)—>D e E
6	D	d	D	(D,d)—>D
7	D	d	Δ	(D,d)—>Δ (symbole vide)
8	*	e		prédiction terminale e identique à l'élément e de la chaine
9	E	c	Δ	(E,c)—>Δ
10	*	#		prédiction terminale # identique à l'élément # de la chaîne

Arbre syntaxique.



Exemple II : Utilisation de règles standards et d'une règle particulière

I	1	2	3	4	5	6	7	8	9	10
CHAINE	#	c	b	d	d	b	d	b	c	#

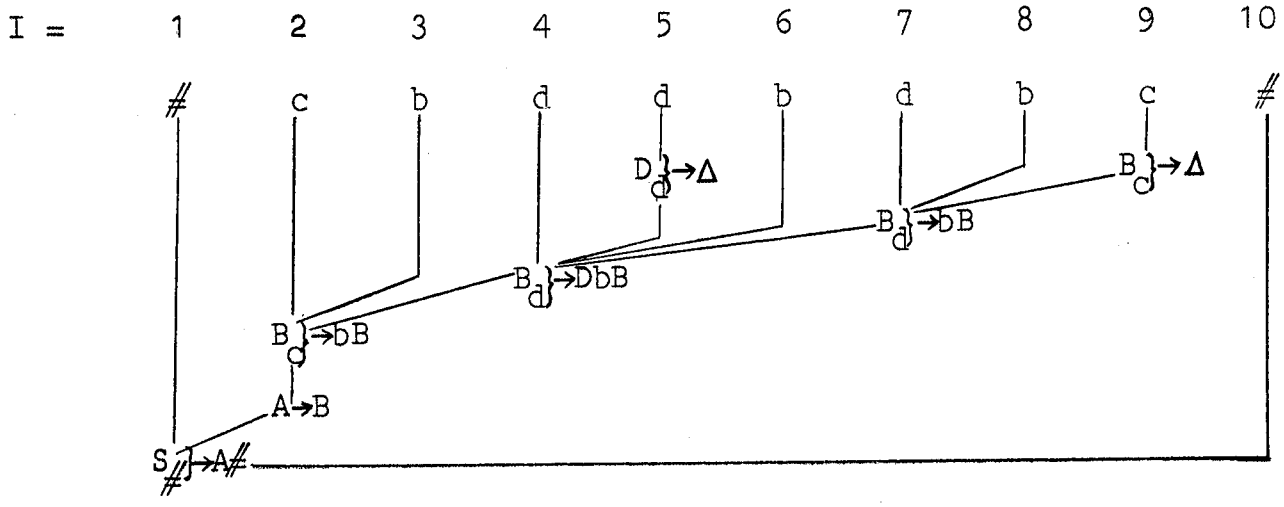
Résultats d'analyse (règles standards)

I	NONTERM	TERMINAL	PREDICTIONS	Commentaires
1	S	#	A #	(S,#) —> A #
2	B	c	b B	(B,c) —> b B
3	*	b		Prédiction terminale b identique à l'élément b de la chaîne
4	B	d	D b B	(B,d) —>D b B
5	D	d	Δ	(D,d) —> Δ
6	*	b		Prédiction terminale b identique à l'élément b de la chaîne
7	B	d	b B	(B,d) —>b B
8	*	b		Prédiction terminale b identique à l'élément b de la chaîne
9	B	c	Δ	(B,c) —> Δ
10	*	#		Prédiction terminale # identique à l'élément # de la chaîne

Résultats particuliers d'analyse (règles particulières)

I	NONTERM PART	PARTIE DROITE	Commentaire
2	A	B	A ⟶ B

Arbre syntaxique



Exemple III : Utilisation de règles standards et de règles particulières.

I	1	2	3	4	5	6	7	8	9	10
CHAINE	#	a	c	f	d	e	c	e	d	#

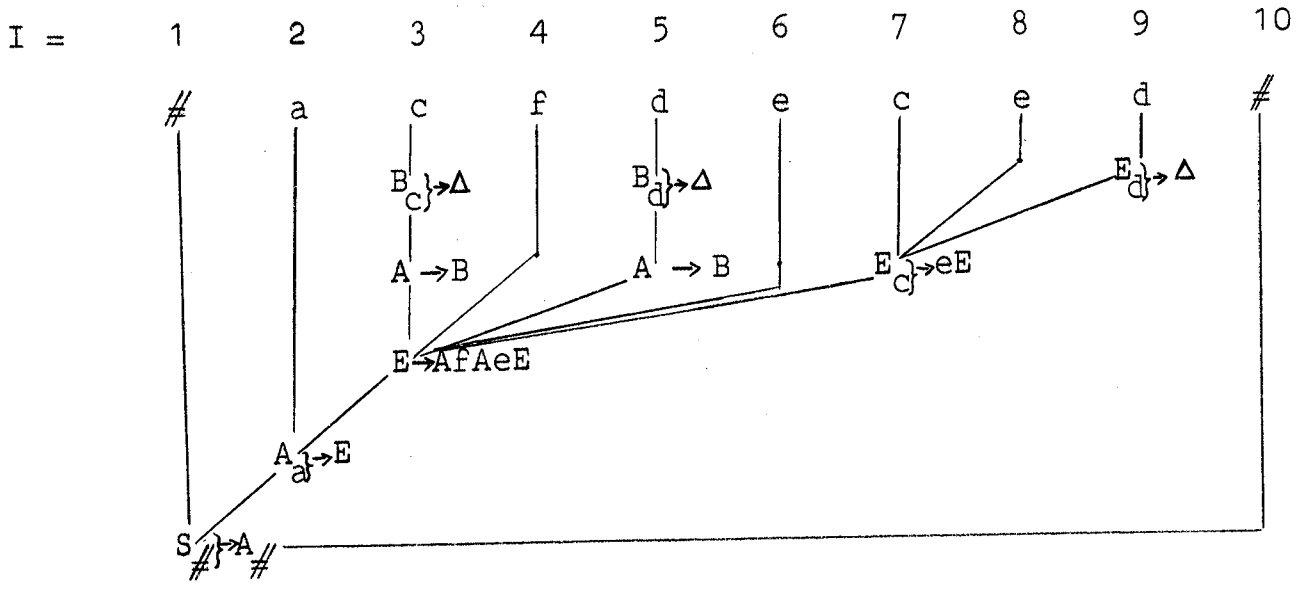
Résultats d'analyse (règles standards)

I	NONTERM	TERMINAL	PREDICTIONS	Commentaires
1	S	#	A #	(S,#) ⟶ A #
2	A	a	E	(A,a) ⟶ E
3	B	c	Δ	(B,c) ⟶ Δ (symbole vide)
4	*	f		Prédiction terminale f identique à l'élément f de la chaîne
5	B	d	Δ	(B,d) ⟶ Δ
6	*	e		Prédiction terminale e identique à l'élément e de la chaîne
7	E	c	e E	(E,c) ⟶ e E
8	*	e		Prédiction terminale e identique à l'élément e de la chaîne
9	E	d	Δ	(E,d) ⟶ Δ
10	*	#		Prédiction terminale # identique à l'élément # de la chaîne

Résultats particuliers d'analyse (règles particulières)

I	NONTERM PART	PARTIE DROITE	Commentaires
3	E	A f A e E	E ⟶ A f A e E
5	A	B	A ⟶ B

Arbre syntaxique.



IV.3. DESCRIPTION DE L'ALGORITHME PERMETTANT UN REORDONNANCEMENT DES REGLES EN FONCTION DE LEUR FREQUENCE D'UTILISATION.

Nous devons d'abord calculer ces fréquences d'utilisation. Ceci est très aisément réalisé à partir des tableaux décrivant la structure de la chaîne donnée. STATISTIQUE, tableau à une dimension d'indice L, contient le nombre d'applications de la règle standard dont l'adresse dans TABLE est TABADRESSE[L]. Le tableau STATISPART joue un rôle analogue en ce qui concerne les règles particulières employées.

Il s'agit ensuite de modifier en conséquence dans TABLE ce que nous avons appelé les données de liaison. La première donnée de liaison d'une "série" de règles (Cf.§ 1) doit prendre la valeur de l'adresse dans TABLE de la règle la plus utilisée... etc. On utilise deux tableaux auxiliaires TABSTA et TABVAL d'indice S : TABSTA[S] contient la fréquence d'utilisation de la règle dont l'adresse dans TABLE est TABVAL[S]. Une fois que, pour une série de règles, leurs éléments sont ordonnés en fonction décroissante de TABSTA[S], il ne reste qu'à affecter à la première donnée de liaison de la série de règles la valeur TABVAL[1] et ainsi de suite.

Nous donnons en premier lieu l'algorithme calculant la fréquence d'emploi des règles et fournissons en exemple le tableau STATISTIQUE obtenu après les analyses successives de trois chaînes satisfaisant à la grammaire G3*. Il s'agit des

chaînes déjà considérées au chapitre précédent.

En second lieu, on trouve l'algorithme de réordonnancement proprement dit. Il est suivi du nouveau tableau TABLE correspondant à la grammaire $G3^{*}$ et modifié par les précédentes observations.

CALCUL DES FREQUENCES D'UTILISATION DES REGLES

```
pour I := 1 pas 1 jusqua IFINAL faire
si NONTERM[I] ≠ ETOILE alors
debut ENT := NONTERM[I] ; J := ADRESSE[I] ;
|     RESULTAT STATISTIQUE :
|         pour L := LMINIMUM[ENT] pas 1 jusqua LMAXIMUM[ENT] faire
|         si J = TABADRESSE[L] alors
|         debut STATISTIQUE[L] := STATISTIQUE[L] +1 ;
|           |     allera SUITE 1
|         fin ;
|     SUITE1 :
fin ;
pour I := 1 pas 1 jusqua IFINAL faire
si NONTERMPART[I] ≠ ETOILE alors
debut
|    si NONTERM PART[I] = NONTERM[I] alors allera SUITE2 ;
|    ENT := NONTERM PART[I] ; J := ADRESSE PART[I] ;
|  RESULTAT STATISTIQUE PARTICULIERE :
|    pour LP := LMINIPART[ENT] pas 1 jusqua LMAXIPART[ENT] faire
|    si J = TABADRPART[LP] alors
|    debut STATISPART[LP] := STATISPART[LP] +1 ;
|      |     allera SUITE2
|    fin ;
|  SUITE2 :
fin ;
```

RESULTATS STATISTIQUES SUR LA FREQUENCE D'UTILISATION CORRECTE DE REGLES NORMALES ET PARTICULIERES D'APRES LES RESULTATS D'ANANALYSE DES 3 EXEMPLES.

Règles normales

	L	STATISTIQUE	Commentaires
S	1	3	(S,#) —> A #
A	2	2	(A,a) —> E
E	3	1	(E,c) —> Δ
	4	1	(E,d) —> Δ
	5	0	(E,d) —> D
	6	2	(E,c) —> e E
	7	0	(E,d) —> e E
	8	1	(E,d) —> DeE
B	9	2	(B,c) —> Δ
	10	1	(B,d) —> Δ
	11	0	(B,d) —> D
	12	1	(B,c) —> b B
	13	1	(B,d) —> b B
	14	1	(B,d) —> DbB
D	15	2	(D,d) —> Δ
	16	1	(D,d) —> D

Règles particulières

	LP	STATISPART	Commentaires
A	1	2	A ⟶ B
E	2	0	E ⟶ A f A
	3	1	E ⟶ A f A e E

ALGORITHME D'ORDONNANCEMENT DES REGLES d'après
LA STATISTIQUE sur les RESULTATS D'ANALYSE

```
pour INT := 1 pas 1 jusqua INTM faire
debut ENT := TABENT[INT] ;
    CLASSEM NORMAL :
     pour IT := 1 pas 1 jusqua ITM faire
     debut ET :=TABET[IT] ; J := REGLE[ENT, ET] ;
          si J = ETOILE alors allera SUITE 60 ;
          si TABLE[J]= 1 alors allera SUITE 60 ;
          S := 0 ;
          pour L :=LMINIMUM[ENT] pas 1 jusqua LMAXIMUM[ENT] faire
          si ET = ELMTERM[L]alors
          debut S := S+1 ; TABSTA[S]:=STATISTIQUE[L] ; TABVAL[S]
                                                :=TABADRESSE[L]
          fin ;
          SM := S ;
          pour S:=1 pas 1 jusqua SM-1 faire
          debut U:=S ;
               CLAS DECROISSANT :
                si TABSTA[U]>TABSTA[U+1] alors allera SUITE 50;
                V:=TABSTA[U]; TABSTA[U]:=TABSTA[U+1] ;TABSTA[U+1] :=V;
                V:=TABVAL[U]; TABVAL[U]:=TABVAL[U+1] ;TABVAL[U+1] :=V;
                U:=U-1 ;
                si U > 0 alors allera CLAS DECROISSANT ;
                SUITE 50 :
          fin de classement décroissant normal ;
```

```
      | CLASPREMREGLE :
      |   J := TABLE[J+1] := TABVAL[1] ;
      | CLAS REG SUIVANTE :
      |   pour S := 2 pas 1 jusqua SM faire
      |   J:=TABLE[J+TABLE[J]+1]:=TABVAL[S] ;
      |   TABLE[J+TABLE[J]+1]:= ETOILE ;
      | SUITE 60 :
   fin de boucle pour IT et de classement des règles normales ;
CLASSEM PARTICULIER :
   J := RAC PART[ENT] ;
   si J = ETOILE alors allera SUITE 80 ;
   si TABLE[J]= 1 alors allera SUITE 80 ;
   S := 0 ;
   pour LP:=LMINIPART[ENT] pas 1 jusqua LMAXIPART[ENT] faire
   debut S:=S+1 ; TABSTA[S] :=STATISPART[LP] ;TABVAL[S] :=
      |                                          :=TABADRPART[LP]
   fin ;
   SM := S ;
   pour S := 1 pas 1 jusqua SM-1 faire
   debut U := S ;
      | CLAS DECR PART :
      |   si TABSTA[U] ⩾ TABSTA[U+1] alors allera SUITE 70 ;
      |   V:=TABSTA[U] ; TABSTA[U] := TABSTA[U+1] ; TABSTA[U+1] :=V;
      |   V:=TABVAL[U] ; TABVAL[U] := TABVAL[U+1] ; TABVAL[U+1] :=V;
      |   U := U-1 ;
      |   si U > 0 alors allera CLAS DECR PART ;
      | SUITE 70 :
   fin de classement décroissant particulier ;
```

```
    CLAS PREM REG PART :
      J := TABLE[J+1] := TABVAL[1] ;
    CLAS REG PART SUIV :
      pour S := 2 pas 1 jusqua SM faire
      J := TABLE[J+TABLE[J]+1]:= TABVAL[S] ;
      TABLE[J+TABLE[J]+1]:= ETOILE ;
    SUITE 80 :
fin de boucle pour INT et de classements normal et particulier ;
```

TABLE modifiée après ORDONNANCEMENT des REGLES

	J	TABLE	Commentaires
(S,#)	(1)	1	NR (nombre de règles)
	2	3	d.l. (donnée de liaison)
	3	2	NE (nombre d'elements)
	4	#	(S,#) → A#
	5	A	
	6	*	Fin des règles (S,#) pas de signification
(A,a)	(7)	1	NR (nombre de règles)
	8	9	d.l.
	9	1	NE
	10	E	(A,a) → E
	11	*	Fin des règles (A,a)
A → règle particulière	12	1	NR
	13	14	d.l.
	14	1	NE
	15	B	A → B
	16	*	Fin des règles particulières A
(E,c)	(17)	2	NR
	18	37	d.l.
	19	1	NE
	20	Δ	(E,c) → Δ
	21	*	Fin des règles (E,c)
(E,d)	(22)	4	NR
	23	24	d.l.
	24	1	
	25	Δ	(E,d) → Δ
	26	45	d.l.
(B,c)	27	1	NE
	28	D	(E,d) → D
	29	41	d.l.
E → règle particulière	30	2	NR
	31	50	d.l.
(B,d)	32	3	NE
	33	A	
	34	f	E → AfA
	35	A	
	36	*	Fin des règles particulières de E
	37	2	NE
	38	E	(E,c) → eE
	39	e	
	40	19	d.l.
	41	2	NE
	42	E	(E,d) → eE
	43	e	
	44	*	Fin des règles (E,d)
	45	3	NE
	46	E	(E,d) → DeE
	47	e	
	48	D	
	49	27	
	50	5	NE
	51	E	
	52	e	
	53	A	E → AfAeE
	54	f	
	55	A	
	56	32	d.l.
(B,c)	57	2	NR
	58	59	d.l.
	59	1	NE
	60	Δ	(B,c) → Δ
	61	70	d.l.
(B,d)	62	4	NR
	63	64	d.l.
	64	1	NE
	65	Δ	(B,d) → Δ
	66	74	d.l.
	67	1	NE
	68	D	(B,d) → D
	69	*	Fin des règles de (B,d)
	70	2	NE
	71	B	(B,c) → bB
	72	b	
	73	*	Fin des règles de (B,c)
	74	2	NE
	75	B	(B,d) → bB
	76	b	
	77	78	d.l.
	78	3	NE
	79	B	
	80	b	(B,d) → DbB
	81	D	
	82	67	d.l.
(D,d)	83	2	NR
	84	85	d.l.
	85	1	NE
	86	Δ	(D,d) → Δ
	87	88	d.l.
	88	1	NE
	89	D	(D,d) → D
	90	*	Fin des règles de (D,d)

IV.4. EXEMPLES DE GRAMMAIRES ET DE CHAINES TRAITEES.

IV.4.1 - Nous avons d'abord procédé à quelques essais d'analyse de chaînes satisfaisant à la grammaire G3*. Le programme est agencé de telle sorte qu'il est possible de réordonner les règles en fonction de leurs fréquences cumulées d'utilisation au cours de l'analyse de plusieurs chaînes. On peut ainsi avoir une meilleure idée de gain de temps réel réalisé en employant cette technique dans l'analyse de chaînes satisfaisant à une grammaire donnée.

Soient $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ les temps d'analyse relatifs aux chaînes :

a c e d d d e c
c b d d b d b c
a c f d e c e d

Nous donnons dans le tableau suivant leurs valeurs respectives dans les trois cas suivants

A. L'ordre des règles est l'ordre initial (Cf.§ 1)
B. Pour chaque chaîne, le nouvel ordre des règles est celui impliqué par une première analyse du type A.
C. L'ordre des règles est celui produit par les analyses successives de type A des trois chaînes.

	A	B	C
$\mathcal{C}_1$	7,2	4,3	4,6
$\mathcal{C}_2$	6,7	4,4	4,6
$\mathcal{C}_3$	9,5	5,2	5,4

Les temps sont donnés en centièmes de seconde.

Les temps obtenus dans le cas B sont évidemment optimaux (du moins pour la technique de réordonnancement utilisée). Nous ne pouvons bien sûr, déterminer un taux d'optimisation : en effet, nous n'avons considéré que les fréquences cumulées correspondant à l'analyse des trois chaînes et ces chaînes sont particulières. Néanmoins, les gains de temps semblent assez substantiels pour que l'on songe à appliquer cette technique à une grammaire proche de celle d'ALGOL.

IV.4.2 - Les essais que nous avons effectués ensuite ont porté sur la grammaire GA déjà présentée au chapitre II ;elle traite principalement les expressions arithmétiques et booléennes.

Les règles que nous avons décidé de ne pas transformer sont celles qui comportent les symboles non terminaux suivants en première position de partie droite :

<expression arithmétique>, <expression arithmétique simple>
<expression arithmétique simple normale>
<expression booléenne simple>

En effet, le remplacement de ces métavariables à l'aide de l'algorithme décrit au chapitre IV.1 engendrerait un nombre considérable de règles standard. Déjà la grammaire équivalente GA^{*} que nous générons à partir de celle-ci (contenant 41 règles) comprend 218 règles standards et 36 particulières.

On a testé les trois programmes suivants :

Programme P1 : identique à celui du chapitre III.1.

Programme P2 :

```
# D:=A[I] + B[I] ;
  F:=si A[I] ⩾ B[I,J] ∨ C[I,J,K] = D ∧ X ∨
     ¬ E ≤ F ⊃ G ≠ H ≡ ¬ (Y ∧ Z) alors 5 sinon A[I];
  T:=I ;
  Z:=3 #
```

Programme P3 :

```
# C:=si si P ∨ Q alors R sinon
     si ∨ ∧ W alors S sinon ¬ U alors B sinon A ;
  D:=C* B+7 ;
  F:=I ;
  I:=5 #
```

Les temps d'analyse $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ obtenus pour ces trois programmes P1, P2, P3 et dans les trois cas A, B, C prévus précédemment sont :

	A	B	C
$\mathcal{C}_1$	164,5	18	23,5
$\mathcal{C}_2$	323,7	5,6	36,7
$\mathcal{C}_3$	10,6	0,2	0,3

temps donnés en secondes

Nous remarquons d'abord que le programme P1 est analysé plus vite par l'algorithme de Kuno-Oettinger que par l'algorithme du chapitre II.1 (Les temps d'analyse comparés sont ceux obtenus dans le cas où on n'a pas tenu compte, dans le

rangement des règles, de leurs parties communes : en effet, la disposition du tableau TABLE ne prévoit pas d'éventuelles parties communes). Ceci est bien conforme à ce que nous attendions. Evidemment, ce gain de temps coûte de la place dans la mémoire de la machine. Les règles de la grammaire transformée sont très détaillées ; ainsi la règle de GA* suivante :

<expression booléenne simple> ::= ¬<primaire booléen> ∧
<facteur booléen> ∨ <terme booléen> ⊃ <implication>
≡ <expression booléenne simple>

Il n'est donc pas étonnant que l'analyse des trois exemples précédents ne nécessite que l'application de 50 règles environ (sur 254) de GA* ; la même analyse avec l'algorithme du chapitre II.1 exigerait l'application de presque toutes les règles de GA.

En second lieu, il est incontestable et ceci était prévisible, qu'un nouvel ordonnancement, dans le cas B, provoque une diminution considérable du temps d'analyse. D'autre part, les résultats obtenus dans le cas C montre que l'amélioration reste très sensible quand le réordonnancement provient des analyses successives des trois programmes. Il est difficile de donner à partir de ces résultats une estimation précise du taux d'optimisation. Il eut fallu pour cela que nous disposions d'un échantillon de programmes qui, analysés successivement, fournissent des fréquences d'utilisation moyennes. On conçoit qu'un tel échantillon soit difficile à déterminer. Il semble que la seule chose que nous puissions affirmer

est que le taux d'optimisation se trouve en général compris entre 2 et 20.

Dans le cas où les fréquences d'utilisation de deux règles d'une même série sont égales, l'ordre de consultation n'est pas modifié. On voit ici qu'un second degré d'optimisation pourrait être envisagé, tenant compte par exemple, de l'ordre dans lequel ces règles ont été appliquées etc...

Enfin, nous devons signaler que l'application d'une règle particulière n'est pas toujours comptabilisée quand plusieurs de ces règles sont nécessaires à la reconnaissance d'un seul symbole terminal. On a seulement retenu l'application de la première utilisée. En fait, nous nous sommes surtout intéressés au rangement des règles standards.

CONCLUSION

Nous avons décrit en détail dans les chapitres précédents quelques algorithmes d'"analyse descendante" examinant des chaînes de langages "context-free". On a étudié en particulier les possibilités pratiques de transformer préalablement une grammaire pour la mettre sous une forme plus propre à une analyse. Danc cette partie, nous discuterons des avantages et inconvénients résultant d'une telle transformation : en conclusion, nous suggèrerons d'autres possibilités d'emploi pour les programmes exposés dans les chapitres précédents.

Il nous semble que l'utilisation de la méthode de Kuno-Oettinger présente deux inconvénients majeurs. Il s'agit de :

1. La disparition de certaines métavariables.

On a montré que toutes les fois qu'une grammaire contient des règles de la forme :

$Z \longrightarrow A\varphi$ $\qquad$ φ est une chaîne sur $V_T \cup V_N \cup \{\Delta\}$

avec A défini seulement par des règles du type

$A \longrightarrow a\psi$ $\qquad$ ψ est une chaîne sur $V_T \cup V_N \cup \{\Delta\}$.

A n'apparaît pas dans la grammaire équivalente obtenue après transformation. Ceci peut provoquer de sérieuses

complications dans la phase de génération d'un compilateur paramétrisé.

Si A est nécessaire au déclenchement d'une action du générateur de code, il faudra reconstruire l'arbre avant de commencer la génération. Le temps passé à ce genre de reconstitution peut très bien surpasser le gain de temps réalisé en utilisant une grammaire transformée. Cependant, pour statuer définitivement sur cette question, il faudrait examiner chaque cas en particulier et les actions correspondantes du générateur. Or, ce problème ne semble pas encore résolu, car on ne connaît (hélas !) que fort peu de choses sur le "contenu sémantique" d'une métavariable. (Signalons tout de même à ce propos que des métavariables sont quelquefois introduites uniquement pour faciliter la description et qu'elles peuvent n'avoir aucun effet sur la génération de code-machine en cours).

2. L'augmentation substantielle du nombre de règles due au choix d'une grammaire transformée.

Un essai de l'algorithme décrit au chapitre IV pour transformer la grammaire ALGOL complète a fourni un ensemble de règles trop important pour pouvoir être rangé dans la mémoire du calculateur. Les 200 règles syntaxiques d'ALGOL se transforment en plusieurs milliers de règles sous forme standard, dont seulement quelques unes seraient couramment utilisées dans une analyse (Cf.Chap.IV.4).

Ce résultat condamne l'utilisation pratique de la

méthode directe de Kuno-Oettinger pour ALGOL. Cependant, ceci n'exclut pas son utilité dans le cas de grammaires particulières et dans les cas où on a choisi une technique de rangement des règles permettant de stocker les parties communes des règles sous forme standard. Kuno dans [Ref.3] a suggéré une telle méthode de rangement (une méthode analogue est employée au chapitre II.1). Elle consiste à introduire un symbole spécial, disons *, qui indique l'existence de "sous-règles" ayant les mêmes constituants initiaux.

Supposons par exemple, qu'une grammaire sous forme standard comprenne les règles suivantes :

$$(A,a) \longrightarrow P\ Q\ R\ S$$
$$(A,a) \longrightarrow R\ S$$
$$(A,a) \longrightarrow \lambda$$

Ces règles sont mises sous la forme d'une règle unique par

$$(A,a) \longrightarrow PQ * RS * \lambda \qquad (1)$$

et l'algorithme les utilise de la façon suivante : chaque fois qu'une de ces règles est nécessaire, la"règle" (1) est placée dans la pile de travail. Si elle se révèle infructueuse, la partie stockée au-dessus du "*" le plus élevé et ce signe lui-même sont effacés de la pile ; on tente alors un nouvel essai. Intuitivement, on se rend compte que l'utilisation de cette sorte de rangement irait quelque peu à l'encontre de l'optimisation obtenue grâce à la technique d'ordonnancement présentée dans le dernier chapitre.

Un autre moyen de remédier aux difficultés occasionnées par le trop grand nombre de règles est d'employer deux modes d'analyse. Le premier serait la méthode normale d'analyse descendante. Une marque spéciale associée à une ou plusieurs métavariables indiquerait qu'on peut utiliser le second mode pour rechercher la métavariable en question ; ce second mode correspondrait à l'analyse de Kuno-Oettinger employant des règles sous forme standard. Par exemple, dans le cas d'ALGOL, ce second mode servirait seulement pour les expressions arithmétiques. Nous utilisons d'ailleurs ce moyen dans l'algorithme présenté au chapitre IV (distinction entre règles standard et règles particulières).

Après avoir examiné les principaux défauts de l'analyseur de Kuno-Oettinger et suggéré quelques procédés pour y remédier, consacrons-nous maintenant à ses avantages qui sont essentiellement :

1) La rapidité accrue de l'analyse.

Déjà intuitivement, on se rendrait compte que l'algorithme d'analyse pour une grammaire standard était plus rapide que l'algorithme classique d'analyse descendante : nous l'avons vérifié pratiquement. Cet accroissement de la vitesse d'analyse est attribuable dans notre cas et dans une certaine mesure à des particularités du calculateur utilisé. Plus exactement, l'instruction TRANSMIT (Cf.IV.1) économise un temps machine important ; elle permet une copie plus rapide de l'information contenue dans la pile de travail dans la

pile de pile (et vice-versa). Ceci montre d'ailleurs qu'on ne peut raisonnablement comparer deux méthodes d'analyse qu'en tenant compte de caractéristiques du calculateur sur lequel elles sont testées.

Nous avons essayé sur l'algorithme décrit au chapitre IV une autre technique augmentant la rapidité de l'analyse. Elle a été aussi suggérée par Kuno [Ref.3] et consiste à abandonner le travail avec un PDS (pile de travail) dès qu'il contient plus d'éléments qu'il ne reste de symboles de base à analyser dans la chaîne d'entrée. Cette technique est susceptible en effet d'économiser du temps machine dans l'analyse des langages naturels où le nombre de mots dans une phrase est relativement faible et où les règles peuvent contenir plusieurs métavariables en partie droite. Dans le cas d'ALGOL, une phrase (ou programme) peut contenir plusieurs milliers de mots, mais la partie droite des règles n'est composée que d'un petit nombre de métavariables. Cette méthode pour accroître la rapidité d'analyse ne semble donc présenter aucun intérêt en ce qui concerne ALGOL. Des essais ont d'ailleurs mis en évidence l'inutilité de cette technique pour des langages du genre ALGOL.

2) La facilité d'adoption du dispositif d'ordonnancement des règles pour augmenter la vitesse d'analyse.

Nous avons montré dans le chapitre précédent que l'ordonnancement des règles qui joue un rôle important dans l'accroissement de la vitesse d'analyse pourrait être très

facilement joint à l'algorihtme de Kuno-Oettinger . On pourraît sans doute trouver des dispositifs analogues pour d'autres analyseurs, mais il n'est pas évident qu'ils seraient aussi efficients et aussi facilement adoptables.

Attachons-nous maintenant dans ce qui suit, à suggérer d'autres utilisations pour les programmes présentés dans ce document.

La première concerne la détection des erreurs. C'est une question qui malheureusement a été souvent négligée par les constructeurs de compilateurs paramétrisés. Pour apprécier la difficulté du problème, rappelons quel est l'effet produit par l'analyse d'une chaîne erronnée (non correcte grammaticalement) à l'aide de l'algorithme donné au chapitre IV. Aucune indication n'est donnée sur l'emplacement possible de l'erreur. La meilleure information s'y rapportant et qui est disponible est le symbole analysé, situé le plus à droite dans la chaîne, information qui peut être évidemment sans grand sens. Une des façons de traiter ce problème est suggérée par IRONS [Ref.5] consiste à introduire des règles auxiliaires qui tiennent compte des fautes les plus courantes. Si une de ces règles a été utilisée pour l'analyse, cela signifie que la chaîne contenait l'erreur correspondante. Cependant, cette méthode présente un inconvénient : l'introduction de ces règles auxiliaires rend souvent le langage ambigu. Dans ce cas, on peut utiliser l'algorithme d'analyse multiple de Kuno-Oettinger pour détecter les erreurs et même

pour les corriger. IRONS [Ref.5] proposait en fait un algorithme de détection d'erreurs qui détermine toutes les structures et qui est très analogue à l'analyseur de Kuno-Oettinger.

Enfin, on peut songer à un autre emploi pour une version modifiée de l'algorithme de transformation décrit dans le chapitre IV. Cet algorithme (sous une autre forme) pourrait être utilisé pour transformer des grammaires "context free" en grammaires d'opérateurs faiblement équivalentes. La caractéristique de ces grammaires d'opérateur est de ne pas admettre deux métavariables adjacentes en partie droite des règles. FLOYD [Ref.6] a démontré qu'un sous-ensemble particulier de ces grammaires d'opérateurs possédaient des propriétés très intéressantes. Ces grammaires définissent des langages qui sont beaucoup plus facilement analysables (du point de vue de rapidité) que les langages "context-free" en général. Un tel programme pourrait être utilisé pour de nouveaux projets de langages.

BIBLIOGRAPHIE

[1] IRONS E.T. - A Syntax Directed Compiler for ALGOL 60 Communications A.C.M. (Janv. 1961)

[2] BROOKER R., MACCALLUM I., MORRIS D., ROHL J. - The Compiler's Compiler. Third Annual Review of Automatic Programming, Pergamon Press, (1963).

[3] HARVARD SUMMER SCHOOL - Language Data Processing, A Special Program, The Computation Laboratory, (Aug.1964).

[4] TAYLOR W., TURNER L., WAYCHOFF R.W. - A Syntactical Chart of ALGOL 60. Communications A.C.M. (Sept. 1961).

[5] IRONS E.T. - An Error Correcting Parse Algorithm, Communications A.C.M. (Nov. 1963).

[6] FLOYD R.W. - Syntactic Analysis and Operator Precedence, Journal A.C.M. (Nov. 1963).

[7] CHEATHAM T., SATTLEY K. - Syntax Directed Compiler, Proceedings Spring Joint Computer Conference, (April 1964).

[8] LEAVENWORTH B.M. - FORTRAN IV as a Syntax Language, Communications A.C.M. (Feb. 1964).

[9] BRASSEUR M., COHEN J. - Algorithmes d'analyse syntaxique pour langages "context free". Revue Chiffres n° 2, (1965).

[10] SAMELSON K. , BAUER F.L. - Sequential formula translation, Communications A.C.M. 3 (Feb. 1960), p.76-83.

[11] KUNO S., OETTINGER A.G. - Multiple Path Syntactic Analyzer, Mathematical Linguistics and Automatic Translation, Harvard University (1963).

[12] GRIFFITHS T.V., PETRICK S.R. - On the Relative Efficiences of Context-Free Grammar Recognizers, Communications A.C.M. (May 1965).

[13] GREIBACH S.- A New Normal Form Theorem for Context-Free Phase Structure Grammars. Journal A.C.M 12 (Jan. 1965),p. 42-52.

[14] GREIBACH S. - Inverses of Phrases Structure Generators. Mathematical Linguistics and Automatic Translation, Report n° NSF 11, Harvard Computation Laboratory (1963).

[15] GREIBACH S. - Formal Parsing System, Communications A.C.M. (Aug. 1964).

[16] FLOYD R.W. - The Syntax of Programming Languages, A Survey IEEE, Transc. EC 13 (Aug. 1964), p. 346-353.

[17] IRONS E.T. - Maintenance Manual for Psyco, Part One, Princeton University, (1961).

[18] WARSHALL S. - A Theorem on Boolean Matrices, Journal A.C.M. n° 1, (1962).

VU,

Grenoble, le

Le Président de la Thèse

VU,

Grenoble, le

Le Doyen de la Faculté des Sciences

VU et permis d'imprimer,

Le Recteur de l'Académie de Grenoble