HAL
open science

Méthode pour la mise au point de grammaire LL(1)

Jérome Bordier

To cite this version:

Jérome Bordier. Méthode pour la mise au point de grammaire LL(1). Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 1971. Français. NNT : . tel-00009479

HAL Id: tel-00009479

https://theses.hal.science/tel-00009479v1

Submitted on 13 Jun 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre

THESE

présentée à

LA FACULTE DES SCIENCES DE L'UNIVERSITE DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR DE TROISIEME CYCLE

" Informatique "

par

Jérôme BORDIER

Méthodes pour la mise au point de grammaires LL(1)

Thèse soutenue le 30.01.71 devant la commission d'examen :

MM	B. VAUQUOIS	Président
	M. BERTHAUD M. GRIFFITHS G. VEILLON	Examinateurs

METHODES POUR LA MISE AU POINT DE GRAMMAIRES LL(1)

LISTE DES PROFESSEURS

Doyen Honoraire : Monsieur M.MORET
Doyen : Monsieur E.BONNIER

PROFESSEURS TITULAIRES

MM.	NEEL Louis	Physique Expérimentale
	KRAVTCHENKO Julien	Mécanique Rationnelle
	CHABAUTY Claude	Calcul différentiel et intégral
	BENOIT Jean	Radioélectricité
	CHENE Marcel	Chimie Papetière
	FELICI Noël	Electrostatique
	KUNTZMANN Jean	Mathématiques Appliquées
	BARBIER Reynold	Géologie Appliquée
	SANTON Lucien	Mécanique des Fluides
	OZENDA Paul	Botanique
	FALLOT Maurice	Physique Industrielle
	KOSZUL Jean-Louis	Mathématiques
	GALVANI Octave	Mathématiques
	MOUSSA André	Chimie Nucléaire
	TRAYNARD Philippe	Chimie Générale
	SOUTIF Michel	Physique Générale
	CRAYA Antoine	Hydrodynamique
	REULOS René	Théorie des Champs
	BESSON Jean	Chimie Minérale
	AYANT Yves	Physique Approfondie
	GALLISSOT François	Mathématiques
Mlle	LUTZ Elisabeth	Mathématiques
MM.	BLANBERT Maurice	Mathématiques
	BOUCHEZ Robert	Physique Nucléaire
	LLIBOUTRY	Géophysique
	MICHEL Robert	Minéralogie et pétrographie
	BONNIER Etienne	Electrochimie et Electrométallurgie
	DESSAUX Georges	Physiologie Animale
	PILLET Emile	Physique Industrielle-Electrotechnique
	YOCCOZ Jean	Physique Nucléaire théorique
	DEBELMAS Jacques	Géologie Générale
	BERBER Robert	Mathématiques
	PAUTENET René	Electrotechnique
	MALGRANGE Bernard	Mathématiques Pures
	VAUQUOIS Bernard	Calcul Electronique
	BARJON Robert	Physique Nucléaire
	BARBIER Jean-Claude	Physique
	SILBERT Robert	Mécanique des Fluides
	BUYLE-BODIN Maurice	Electronique
	DREYFUS Bernard	Termodynamique

MM.	KLEIN Joseph	Mathématiques
	VAILLANT François	Zoologie et Hydrobiologie
	ARNAUD Paul	Chimie
	SENGEL Philippe	Zoologie
	BARNOUD Fernand	Biosynthèse de la Cellulose
	BRISSONNEAU Pierre	Physique
	GAGNAIRE Didier	Chimie Physique
Mme	KOFLER Lucie	Botanique
MM.	DEGRANGE Charles	Zoologie
	PEBAY-PEROULA Jean Claude	Physique
	RASSAT André	Chimie Systématique
	DUCROS Pierre	Cristallographie Physique
	DODU Jacques	Mécanique Appliquée I.U.T.
	ANGLES D'AURIAC Paul	Mécanique des Fluides
	LACAZE Albert	Thermodynamique
	GASTINEL Noël	Analyse Numérique
	GIRAUD Pierre	Géologie
	PERRET René	Servo-mécanisme
	PAYAN Jean Jacques	Mathématiques Pures

PROFESSEURS SANS CHAIRE

MM.	GIDON Paul	Géologie
Mme	BARBIER Marie-Jeanne	Electrochimie
Mme	SOUTIF Jeanne	Physique
MM.	COHEN Joseph	Electrotechnique
	DEPASSEL R.	Mécanique des Fluides
	GLENAT René	Chimie
	BARRA Jean	Mathématiques Appliquées
	COUMES André	Electronique
	PERRIAUX Jacques	Géologie et Minéralogie
	ROBERT André	Chimie Papetière
	BIARREZ Jean	Mécanique Physique
	BONNET Georges	Electronique
	CAUQUIS Georges	Chimie Générale
	BONNETAIN Lucien	Chimie Minérale
	DEPOMMIER Pierre	Physique Nucléaire-Génie Atomique
	HACQUES Gérard	Calcul Numérique
	POLOUJADOFF Michel	Electrotechnique
Mme	KAHANE Josette	Physique
Mme	BONNIER Jane	Chimie
MM.	VALENTIN Jacques	Physique
	REBECQ Jacques	Biologie
	DEPORTES Charles	Chimie
	SARROT-REYNAULD Jean	Géologie
	BERTRANDIAS Jean Paul	Mathématiques
	AUBERT Guy	Physique

PROFESSEURS ASSOCIES

MM.	RODRIGUES Alexandre	Mathématiques Pures
	MORITA Susumu	Physique Nucléaire
	RADHAKRISHNA	Thermodynamique

MAITRES DE CONFERENCES

MM.	LANCIA Roland	Physique Atomique
Mme	BOUCHE Liane	Mathématiques
MM.	KAHANE André	Physique Générale
	DOLIQUE Jean Michel	Electronique
	BRIERE Georges	Pyysique
	DESRE Georges	Chimie
	LAJZEHOWICZ Joseph	Physique
	LAURENT Pierre	Mathématiques Appliquées
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	LONGEQUEUE Jean Pierre	Physique
	SOHM Jean Claude	Electrochimie
	ZADWORNY François	Electronique
	DURAND Françis	Chimie Physique
	CARLIER Georges	Biologie Végétale
	PFISTER Jean Claude	Physique
	CHIBON Pierre	Biologie Animale
	IDELMAN Simon	Physiologie animale
	BLOCH Daniel	Electrotechnique I.P.
	MARTIN-BOUYER Michel	Chimie (C.S.U.Chambery)
	SIBILLE Robert	Construction mécanique (I.U.T.)
	BRUGEL Lucien	Energétique I.U.T.
	BOUVARD Maurice	Hydrologie
	RICHARD Lucien	Botanique
	PELMONT Jean	Physiologie animale
	BOUSSARD Jean Claude	Mathématiques Appliquées(I.P.G.)
	MOREAU René	Hydraulique I.P.G.
	ARMAND Yves	Chimie I.U.T.
	BOLLIET Louis	Informatique I.U.T.
	KUHN Gérard	Energétique I.U.T.
	PEFFEN René	Chimie I.U.T.
	GERMAIN Jean Pierre	Mécanique
	JOLY Jean René	Mathématiques Pures

Mlle	PIERY Yvette	Biologie Animale
MM.	BERNARD Alain	Mathématiques Pures
	MOHSEN Tahsin	Biologie (C.S.U.Chambery)
	CONTE René	Mesures Physiques I.U.T.
	LE JUNTER Noël	Génie Electrique Electronique I.U.T.
	LE ROY Philippe	Génie Mécanique I.U.T.
	ROMIER Guy	Technique Statistiques Quantitatives I.U.T
	VIALON Pierre	Géologie
	BENZAKEN Claude	Mathématiques Appliquées
	MAYNARD Roger	Physique
	DUSSAUD René	Mathématiques (C.S.U.Chambery)
	BELORIZKY Elie	Physique (C.S.U Chambery)
Mme	LAJZEROWICZ Jeanine	Physique (C.S.U Chambery)
M.	JULLIEN Pierre	Mathématiques Pures
Mme	RINAUDO Marguerite	Chimie
MM.	BLIMAN Samuel	E.I.E.
	BEGUIN Claude	Chimie Organique
	NEGRE Robert	I.U.T.

MAITRES DE CONFERENCES ASSOCIES

MM.	YAMADA Osamu	Physique du Solide
	NAGAO Makoto	Mathématiques Appliquées
	MAREZIO Massimo	Physique du Solide
	CHEECKE John	Thermodynamique
	BOUDOURIS Georges	Radioélectricité
	ROZMARIN Georges	Chimie Papetière

Je tiens à remercier

Monsieur B. VAUQUOIS, *Directeur du Centre d'Etude pour la Traduction Automatique, qui me fait l'honneur de présider le jury,*

Monsieur M. BERTHAUD, *Ingénieur au Centre Scientifique IBM France, avec qui j'ai eu de nombreuses et passionnantes discussions,*

Monsieur M. GRIFFITHS, *Maître de Conférences à la Faculté des Sciences de Grenoble, qui m'a donné mon sujet et m'a dirigé "en temps réel",*

Monsieur G. VEILLON, *Maître de Conférences à l'Institut Polytechnique de Grenoble qui s'est intéressé à mes recherches.*

Je remercie tout spécialement Monsieur M. PELTIER, *Directeur du Centre Scientifique IBM France, qui ne m'a pas ménagé ses encouragements et ses conseils et Monsieur* J. C. BOUSSARD, *Maître de Conférences à l'Institut Polytechnique qui m'a fait découvrir l'analyse syntaxique et m'a suivi et conseillé dans mon travail.*

Je remercie particulièrement mes collègues Messieurs P.Y. CUNIN, A. LANDELLE, M. LEVY, J. PLEYBER, H. SAYA, M. SIMONET *qui, par leur collaboration et leur aide, ont contribué à la réalisation de ma thèse.*

Un grand merci à Mademoiselle J. QUACCHIA, *pour sa dactylographie exécutée dans des conditions souvent difficiles.*

Je remercie également Messieurs C. ANGUILLE, C. LABORIE, *et* P. MOUNET, *pour leur travail de tirage.*

TABLE DES MATIERES

TABLE DES MATIERES

	Pages
TABLE DES MATIERES	I
INTRODUCTION	3
CHAPITRE I : NOTATIONS ET DEFINITIONS	6
1.1. Généralités	6
1.2. Systèmes d'équations	7
1.3. Relations de dépendance	8
1.4. Graphes	10
1.5. Grammaires LL(1), LL(k). S-grammaires	12
1.6. Définition de la notation ULD	13
CHAPITRE II : TRANSFORMATIONS ELEMENTAIRES	16
2.1. Définition	16
2.2. Elimination de la récursivité à gauche	16
2.2.1. Graphe associé à un système d'équations	17
2.2.2. Elimination de la récursivité à gauche	20
2.2.3. Algorithme de transformation	23
2.2.4. Cas des grammaires possédant des règles du type $A \to \varepsilon$ ou $A \to B$	28
2.2.5. Fonctions sémantiques	35
2.3. Factorisation à gauche	37
2.3.1. Définition	37
2.3.2. Propriété	38
2.3.3. Propriétés	38
2.3.4. Cas des fonctions sémantiques	40

2.4. Substitution à gauche 41

2.4.1. Définition 41
2.4.2. Propriétés 41
2.4.3. Fonctions sémantiques 42

2.5. Factorisation à droite 43

2.5.1. Définition 43
2.5.2. Propriété 43
2.5.3. Propriétés 43
2.5.4. Fonctions sémantiques 44

CHAPITRE III : METHODES DE TRANSFORMATION 46

3.1. Introduction 46
3.2. Méthode de substitution-factorisation (SID) 48

3.2.1. Algorithme de numérotation 49
3.2.2. Elimination de la récursivité à gauche 49
3.2.3. Première factorisation et calcul de ℓ^+ 49
3.2.4. Substitution-factorisation 50
3.2.5. Fonctions sémantiques 53
3.2.6. Remarque 53
3.2.7. Etude de la méthode SID 53

3.3. Méthode de Paull et Unger 58
3.4. Méthodes particulières 62

3.4.1. Cas de règles qui ne satisfont pas à la condition LL(1) n° 3 62
3.4.2. Cas de règles ne satisfaisant pas à la condition LL(1) n° 2 64

CHAPITRE IV : ETUDE DE LA NOTATION ULD 67

4.1. Introduction 67
4.2. Définitions 67
4.3. Cas des règles récursives 69
4.4. Avantages de la notation ULD 70
4.5. Traduction de ULD en BNF 71

4.6. Conditions LL(1) pour une grammaire ULD 73
4.7. Programme de traduction 74
4.8. Génération d'un analyseur directement à partir d'ULD 76

CHAPITRE V : PROGRAMMES 79

5.1. Les programmes de transformation de grammaire 79
5.2. Outils des programmes 81
5.2.1. Gestion de listes 81
5.2.2. Représentation de la grammaire 84
5.2.3. Calcul des relations fondamentales ℓ^+ et s 84
5.3. Forme de la grammaire d'entrée 84
5.3.1. Programme BNF 84
5.3.2. Programme ULD 85

CHAPITRE VI : APPLICATIONS 88

6.1. Programme REC (élimination de la récursivité à gauche) 88
6.2. Programme ULD. Application à la syntaxe de PL/1 90
6.2.1. Introduction 90
6.2.2. Résultats 91
6.2.3. Etude des règles qui ne sont pas LL(1) 92
6.2.4. Remarques 95

CONCLUSION 98

APPENDICE A Exemple d'analyseur 101

APPENDICE B Définition des grammaires LL(k) 104

APPENDICE C Hiérarchie de langages déterministes 106
Propriétés de fermeture 107
Décidabilité 108

BIBLIOGRAPHIE 111

Figure 1 11
Figure 2 19
Figure 3 26
Figure 4 80

INTRODUCTION

INTRODUCTION

L'analyse descendante déterministe est peut-être actuellement la plus simple et la plus rapide pour les langages de programmation. Ce type d'analyse a conduit à la caractérisation des grammaires LL(k), $k \geq 1$ [Lew] et a été formalisé par D.E. Knuth pour $k = 1$ [Kn2].

Monsieur Griffiths et Monsieur Peltier ont réalisé un programme traduisant une grammaire LL(1) en un analyseur écrit en macro-instructions et exécutable sur IBM 360. Ce programme transforme la grammaire d'entrée, puis examine si la grammaire obtenue est LL(1). Si elle l'est alors l'analyseur est généré.

Le type d'analyse choisi imposant des contraintes à la grammaire, il est nécessaire d'écrire (ou de réécrire) la grammaire d'un langage de programmation de manière à satisfaire les conditions LL(1). On est amené au problème suivant : étant donnée une grammaire context-free quelconque, comment faire pour la transformer en une grammaire LL(1) ? et à la question suivante : étant donnée une grammaire context-free quelconque, existe-t-il ou non une grammaire LL(1) équivalente ?

L'indécidabilité de la deuxième question [Ros2] n'empêche pas de s'attaquer à la première. En effet la plupart des transformations ne sont pas effectuées par le programme ci-dessus. Il faut les réaliser à la main, ce qui est souvent long et compliqué. L'étude et la réalisation d'algorithmes de transformations répond à cette difficulté.

Le présent travail consiste donc, à partir des méthodes de différents auteurs (en particulier J.M. Foster [Fos]), en la définition de ces transformations, l'examen de leurs propriétés et enfin les améliorations qu'on peut apporter à ces méthodes.

L'étude de la notation ULD, qui est une forme normale de Backus étendue, montrera un autre moyen de faciliter l'écriture d'une grammaire LL(1).

Cette dernière étude, ainsi que celle de l'élimination de la récursivité à gauche dans une grammaire context-free doivent être publiées [Bor1, Bor2].

Nous avons écrit des programmes réalisant certaines transformations : élimination de la récursivité à gauche, substitutions et factorisations à gauche. Un autre programme traduit une grammaire ULD en une grammaire sous forme normale de Backus. Ils ont été appliqués à une grammaire context-free du langage ALGOL 68 et à une grammaire ULD du langage PL/1.

CHAPITRE I

NOTATIONS ET DEFINITIONS

CHAPITRE I

NOTATIONS ET DEFINITIONS

1. NOTATIONS ET DEFINITIONS

1.0. Pour l'étude des grammaires context-free, on dispose d'outils mathématiques puissants et variés. On peut associer à une telle grammaire un système d'équations, un ensemble de relations représentables par des matrices booléennes, un ou plusieurs graphes orientés. On peut ainsi utiliser des résultats théoriques qui facilitent l'étude envisagée. Avant de voir de quelle façon on peut introduire ces outils, on définit les notations généralement utilisées dans la suite de cet ouvrage. Ensuite on donne la définition d'une grammaire LL(1), puis d'une forme normale de Backus étendue, appelée notation ULD.

1.1. Généralités

Les notations et définitions générales utilisées sont celles rencontrées le plus souvent dans la théorie des langages. Elles sont brièvement rappelées ici.

Etant donné un vocabulaire V_T, ensemble fini de symboles, V_T^* représente l'ensemble de toutes les chaînes formées sur V_T par concaténation (monoïde libre). Un langage est un sous-ensemble de V_T^*.

Une grammaire context-free (C.F.) ou hors-contexte est définie par :

- un ensemble fini de symboles terminaux V_T
- un ensemble fini de symboles non-terminaux (ou auxiliaires) V_N disjoint de V_T
- un symbole distingué S, appartenant à V_N, appelé axiome

- un ensemble fini de <u>règles</u> de la forme :

$$A \rightarrow \psi \qquad (1)$$

$\alpha \overset{+}{\rightarrow} \beta$ indique une <u>dérivation</u>.

$\alpha \overset{L+}{\rightarrow} \beta$ indique une <u>dérivation gauche</u> (à chaque pas de la dérivation, on remplace le non-terminal le plus à gauche).

$\alpha \overset{*}{\rightarrow} \beta$ signifie $\alpha \overset{+}{\rightarrow} \beta$ ou $\alpha = \beta$.

Le <u>langage</u> généré par une grammaire G est l'ensemble

$$L(G) = \{x : S \overset{+}{\rightarrow} x\}$$

La <u>longueur</u> de la chaîne α est notée $\ell(\alpha)$.
$m(\alpha)$ est la longueur de la plus petite chaîne terminale dérivable à partir de α. La chaîne de longueur nulle (ou chaîne vide) est notée ε (élément neutre pour la concaténation). On voit que si $\alpha \overset{+}{\rightarrow} \varepsilon$ alors $m(\alpha) = 0$.

On peut écrire l'ensemble des règles d'une grammaire CF en utilisant une forme identique à la forme normale de Backus (BNF) où le metasymbole '→' remplace '::=' soit :

$$A \rightarrow \psi_1 \mid \psi_2 \mid \ldots \mid \psi_n$$

1.2. Systèmes d'équations

Un langage CF peut être défini par un système d'équations dans l'ensemble des parties de V_T^* (c'est-à-dire des langages). Cet ensemble a une structure algébrique appelée 'gerbier libre' et comporte deux lois de compositions internes :

---

(1) Sauf indication contraire, on désigne par :
a, b, c, d, e, f, g, h des éléments de V_T
u, v, w, x, y, z des chaînes sur V_T (appelées 'chaînes terminales')
A, B, C, ..., X, Y, Z des éléments de V_N
α, β, γ, ...,ω des chaînes sur V, union de V_N et de V_T

- réunion ensembliste de chaînes, notée + (l'élément neutre est l'ensemble vide de chaînes, noté φ).
- concaténation, comme d'habitude sans notation opératoire.

Le système d'équations s'obtient directement à partir des règles BNF. On associe à chaque A, élément de V_N, l''expression polynomiale' $\psi_1 + \psi_2 + \ldots + \psi_n$ et l'équation :

$$A = \psi_1 + \psi_2 + \ldots + \psi_n$$

où les symboles auxiliaires sont considérés comme des 'variables' prenant leurs 'valeurs' dans l'ensemble des langages.

Pour l'étude de ce formalisme les références sont [Bou, Cho, Gin1, 2 et 3, Len]. Une théorie plus générale, celle des séries formelles a été développée [Sch, Len].

1.3. Relations de dépendance

Etant donnée une grammaire CF, on définit des relations binaires sur le vocabulaire V.

Dans ces définitions, X, Y, U, V représentent des symboles terminaux ou non-terminaux, $n \geq 0$ et $A_i \overset{*}{\rightarrow} \varepsilon$ pour $i = 1, 2, \ldots, n$.

1.3.1. Relation de dépendance : d

XdY si il existe une règle $X \rightarrow \psi_1 Y \psi_2$

1.3.2. Relation de dépendance à gauche apparente : ℓ_a

$X \ell_a Y$ si il existe une règle $X \rightarrow Y\psi$

1.3.3. Relation de dépendance à gauche : ℓ

$X \ell Y$ si il existe une règle $X \rightarrow A_1 A_2 \ldots A_n Y \psi$

1.3.4. Relation de dépendance à droite : r

XrY si il existe une règle $X \rightarrow \psi Y A_1 A_2 \ldots A_n$

1.3.5. Relation de succession :

XsY si il existe une règle $W \rightarrow \psi_1 U A_1 A_2 \ldots A_n V \psi_2$
et si les relations
$Ur^* X$ et $V\ell^* Y$ sont vérifiées.

Pour toute relation binaire b, b^+ est la fermeture transitive de cette relation et b^* sa fermeture transitive réflexive $b^* = b^\circ \cup b^+$

Remarque

Si il n'y a pas de règle $A \rightarrow \varepsilon$ dans la grammaire, alors les relations ℓ et ℓ_a sont identiques.

1.3.6. A l'aide des relations définies, on associe à chaque élément de V_N des sous-ensembles (éventuellement vides) de V_T :

premier (A) = $\{a : A\ell^+ a\}$
dernier (A) = $\{a : Ar^+ a\}$
suivant (A) = $\{a : Asa\}$

Pour toute chaîne ψ sur le vocabulaire V

$$\psi = X_1 X_2 \ldots X_m, \; m > 0$$

soit :

$t(\psi) = X_1$
$q(\psi) =$ si $m = 1$ alors ε sinon si $m > 1$ alors $X_2 X_3 \ldots X_m$

On définira récursivement un autre sous-ensemble très important :

Premier (ψ)

Premier (ψ) = si $\psi = \varepsilon$ alors $\emptyset$ sinon
si $t(\psi)$ appartient à V_T alors $t(\psi)$
sinon si $t(\psi) \overset{+}{\rightarrow} \varepsilon$
alors premier ($t(\psi)$) . Premier ($q(\psi)$)
sinon premier ($t(\psi)$)

Premier (ψ) est l'ensemble des symboles terminaux qui peuvent commencer les chaînes terminales dérivables à partir de ψ :

$$\text{Premier}(\psi) = \{t(x) : \psi \overset{*}{\rightarrow} x\}$$

1.3.7. Eléments remarquables de V_N

Tout symbole non-terminal A qui ne vérifie pas la relation Sd^+A est dit inaccessible.

Tout symbole non-terminal A, qui ne dérive pas sur au moins une chaîne terminale x, est dit parasite.

Une grammaire CF peut être réduite en une grammaire équivalente ne possédant ni symboles inaccessibles, ni symboles parasites [Mar].

Tout symbole non-terminal A vérifiant respectivement

$$Ad^+A,\ A\ell^+A$$

est dit récursif, récursif à gauche.

1.3.8. Calcul des relations

Les relations les plus importantes sont ici ℓ, ℓ^+, s. On les représentera par des matrices booléennes et on effectuera les calculs sur ces matrices.

Pour calculer ℓ^+ et s on utilise les algorithmes décrits dans [Gri2].

On peut aussi utiliser l'algorithme de Warshall pour la fermeture transitive. [War].

1.4. Graphes

A chaque relation binaire (d par exemple) correspond un graphe orienté. A un élément du vocabulaire on associe un sommet du graphe. Pour simplifier, on désignera un sommet par le symbole associé.

Si la relation XdY est vraie, alors un arc orienté relie le sommet X au sommet Y. (voir Fig. 1).

On appelle <u>chemin</u> toute suite $X_1, X_2, \ldots, X_n$ de sommets telle qu'il existe un arc de X_i à X_{i+1} pour $i = 1, 2, \ldots, n - 1$.

Un <u>circuit</u> est un chemin fermé (c'est-à-dire $X_1 = X_n$). La longueur d'un chemin est le nombre d'arcs qui le composent. Une <u>boucle</u> est un circuit de longueur 1.

Dans le graphe de la relation d, s'il y a des circuits il y a des symboles non-terminaux récursifs et réciproquement.

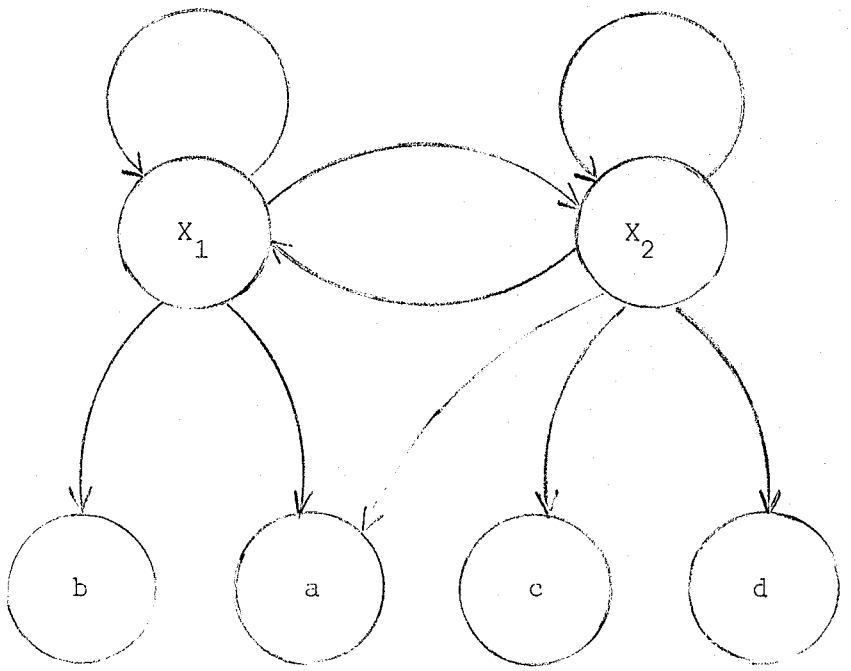
Si ce graphe n'est pas connexe alors il y a un ou plusieurs symboles inacessibles dans la grammaire et réciproquement.

grammaire G_1 $\quad V_T = \{a, b, c, d\}$

$V_N = \{X_1, X_2\}$

$X_1 \rightarrow X_1aX_2 \mid X_2X_2 \mid b$

$X_2 \rightarrow X_2d \mid X_2X_1a \mid aX_1 \mid c.$



<u>Graphe de la relation d.</u>

<u>FIG. 1</u>

1.5. Grammaires LL(1), LL(k). S-grammaires

1.5.1. Définition

Pour qu'une grammaire soit LL(1), il faut et il suffit qu'elle vérifie les conditions suivantes pour toute règle

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

1°) A n'est pas récursif à gauche

2°) Premier $(\alpha_i) \cap$ Premier $(\alpha_j) = \emptyset$ pour tout i, j, $i \neq j$

3°) si il existe j tel que $\alpha_j \overset{*}{\to} \varepsilon$ alors
Premier $(\alpha_i) \cap$ suivant $(A) = \emptyset$ pour tout i,

4°) au plus un α_i dérive sur la chaîne vide.

On les désignera dans la suite par 'conditions LL(1) numéro 1, 2, 3, 4'.

Ces conditions sont telles que les a formulées D.E. Knuth [Kn2]. Une caractérisation différente est celle de M. Griffiths [Gri4]. La définition de grammaire LF [Wo2] est aussi équivalente à LL(1). (LF = left factored).

On appellera 'langage LL(1)' un langage pour lequel il existe une grammaire LL(1).

1.5.2. Analyse descendante déterministe

Les langages LL(1) sont analysables par une méthode descendante, déterministe. Cette analyse est fondée sur un schéma prédictif tel qu'on lit la chaîne d'entrée de gauche à droite en regardant un seul symbole en avant, la connaissance de ce symbole déterminant de manière unique l'action à effectuer. On donne en appendice A, sur un exemple simple, le fonctionnement de l'analyseur.

Quand l'action dépend de la lecture de k caractères en avant, pour k fixé, le langage reconnu est dit LL(k) $k \geq 1$. Voir la définition en appendice B.

On donnera à la fin (appendice C) une vue d'ensemble sur les résultats connus sur ces langages, ainsi que les S-langages définis ci-dessous.

1.5.3. S-grammaires et S-langages

Les S-langages (langages déterministes simples) sont les langages reconnus par une famille d'automates à pile déterministes particuliers [Kor].

Une S-grammaire est telle que pour toute règle

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

1°) $\alpha_i \neq \varepsilon$ quel que soit i

2°) $t(\alpha_i)$ est un symbole terminal

3°) les symboles $t(\alpha_i)$ sont tous différents.

Une S-grammaire génère un S-langage et inversement tout S-langage peut être généré par une S-grammaire.

L'ensemble des S-langages est strictement contenu dans celui des langages LL(1).

1.6. Définition de la notation ULD

La notation ULD est une forme de Backus étendue utilisée pour définir une grammaire du langage PL/1 par les laboratoires IBM de Hursley et de Vienne. [Urs]. Cette notation utilise, outre '::=', '|' et le signe de concaténation implicite, les métasymboles '{', '}', '[', ']', '.' et '...'

Afin de faciliter l'écriture, on représentera ces métasymboles par un autre jeu de caractères, soit respectivement :

→ | < > ( ) . +

Quand il y aura confusion possible avec la représentation d'un symbole terminal, ce dernier sera souligné.

La signification de ces métasymboles est donnée par les exemples qui suivent.

1.6.1.

goto-statement → GOTO references; | GO TO reference ;

est équivalent à :

goto-statement → < GOTO | GO TO > reference ;

1.6.2.

return-statement → RETURN; | RETURN (expression) ;

est équivalent à :

return-statement → RETURN ((expression)) ;

1.6.3.

integer → digit | integer digit

est équivalent à :

integer → digit +

1.6.4.

declarationlist → declaration (< , declaration > +)

est équivalent à :

declarationlist → < . , . declaration . >

Ici les mots en majuscules sont des symboles terminaux et les mots en minuscules (avec ou sans tiret) sont des symboles non-terminaux. On a modifié la notation pour l'expression des listes (1.6.4) car la notation de listes originale utilise le métasymbole ... (soit +) déjà utilisé dans 1.6.3 (distinction peu facile entre les deux utilisations).

Cette notation ULD fera l'objet d'une étude approfondie au chapitre 4.

CHAPITRE II

TRANSFORMATIONS ELEMENTAIRES

CHAPITRE II

TRANSFORMATIONS ELEMENTAIRES

2. TRANSFORMATIONS ELEMENTAIRES

2.1. Définition

Nous entendons par 'transformation d'une grammaire CF', étant donnée une grammaire CF G, un procédé destiné à obtenir une grammaire équivalente G' qui satisfasse certaines conditions (les conditions LL(1)). 'Grammaire équivalente' veut dire que G' génère le même langage que G (on dit aussi équivalence faible [Bous]). Le mot transformation n'a pas le même sens ici que dans la théorie des grammaires transformationnelles de Chomsky [Cho p. 296]. Mais nous l'avons choisi plutôt que le terme réduction (utilisé par exemple pour supprimer les règles $A \to \varepsilon$ dans une grammaire CF) parce que les procédés étudiés sont destinés à obtenir un type bien particulier de grammaire, une grammaire LL(1). On verra aussi que ces procédés ne conservent pas en général la structure des chaînes du langage.

Les transformations élémentaires seront les outils des algorithmes étudiés au chapitre 3. On étudie dans ce chapitre l'élimination de la récursivité à gauche, la factorisation à gauche, la substitution à gauche, et la factorisation à droite. Pour chacune de ces transformations, on envisage le cas où des fonctions sémantiques sont incluses dans la grammaire.

2.2. Elimination de la récursivité à gauche

La première condition LL(1) interdit la récursivité à gauche. Remarquons que cette restriction n'est pas propre aux langages LL(1), mais concerne les algorithmes d'analyse descendante généraux [Grif]. En analyse ascendante au contraire cette condition n'existe pas.

Une grammaire CF peut toujours être réduite à une grammaire équivalente sans récursivité à gauche. [Gre2, Kur1].

L'algorithme de Foster, que l'on va étudier, effectue cette élimination. Dans le cas le plus simple, d'une règle :

$$X \to Xb \mid a$$

on obtient deux nouvelles règles :

$$X \to aY$$
$$Y \to \varepsilon \mid bY$$

Cette transformation, dans sa forme la plus générale, est l'application d'une méthode utilisée pour mettre une grammaire sous forme normale. A partir de cette remarque (§ 2.2.1), on montre que l'algorithme de Foster élimine la récursivité à gauche dans une grammaire ne possédant pas de règles du type (i) ou (ii) (§ 2.2.2).

(i) $X_j \to \varepsilon$

(ii) $X_j \to X_k$

(pour toute grammaire CF, on peut se ramener à ce cas [Bous]).

Ensuite (§ 2.2.3) une propriété tirée de la théorie des graphes permettra de décrire l'algorithme sous sa forme la plus simple. Mais si on accepte des règles du type (i) ou (ii), l'algorithme n'élimine pas toujours la récursivité à gauche. On en donnera les raisons dans le paragraphe 2.2.4.

2.2.1. Graphe associé à un système d'équations

On adopte la définition d'un langage CF par un système d'équations. Soit $V_N = \{X_1, \ldots, X_n\}$, X_1 l'axiome. Chaque équation a la forme générale :

(SO) $X_i = f_i(X_1, \ldots, X_n)$ pour $i = 1, \ldots, n$

f_i étant une expression polynomiale des variables $X_1, \ldots, X_n$.
Le système (SO) peut toujours être mis sous l'une ou l'autre des formes suivantes :

(S1) $X_i = X_1\beta_{1i} + \ldots + X_n\beta_{ni} + \alpha_i$

(S2) $X_i = \beta'_{1i}X_1 + \ldots + \beta'_{ni}X_n + \alpha'_i$

β_{ji}, α_i, β'_{ji}, α'_i représentent des chaînes ou des sommes de chaînes formées sur le vocabulaire total V, qui peuvent être égales à φ, mais jamais à ε. (1)

Toute chaîne composant $\alpha_i(\alpha'_i)$ a comme premier symbole à gauche (dernier symbole à droite) un terminal.

On obtient pour chaque système (S1) ou (S2) un graphe orienté :

- les sommets associés respectivement à $X_1, \ldots, X_n$ plus un sommet T appelé terminal.
- pour tout $\beta_{ji} \neq \varphi$ il existe un arc orienté de X_i vers X_j portant l'étiquette β_{ji}
- pour tout $\alpha_i \neq \varphi$ il existe un arc orienté de X_i vers T portant l'étiquette α_i

(mêmes conventions pour β'_{ji} et α'_i).

<u>(S1) donne le graphe G1 relatif à la relation de dépendance gauche ℓ.</u>
<u>(S2) donne le graphe G2 relatif à la relation de dépendance droite r.</u>

En effet $\beta_{ji} \neq \varphi$ si et seulement si $X_i \ell X_j$ est vrai
$\beta'_{ji} \neq \varphi$ si et seulement si $X_i r X_j$ est vrai (1)

et $\alpha_i \neq \varphi$ si et seulement si il existe a tel que $X_i \ell a$ est vrai (même énoncé avec α'_i et r), a est un symbole terminal.

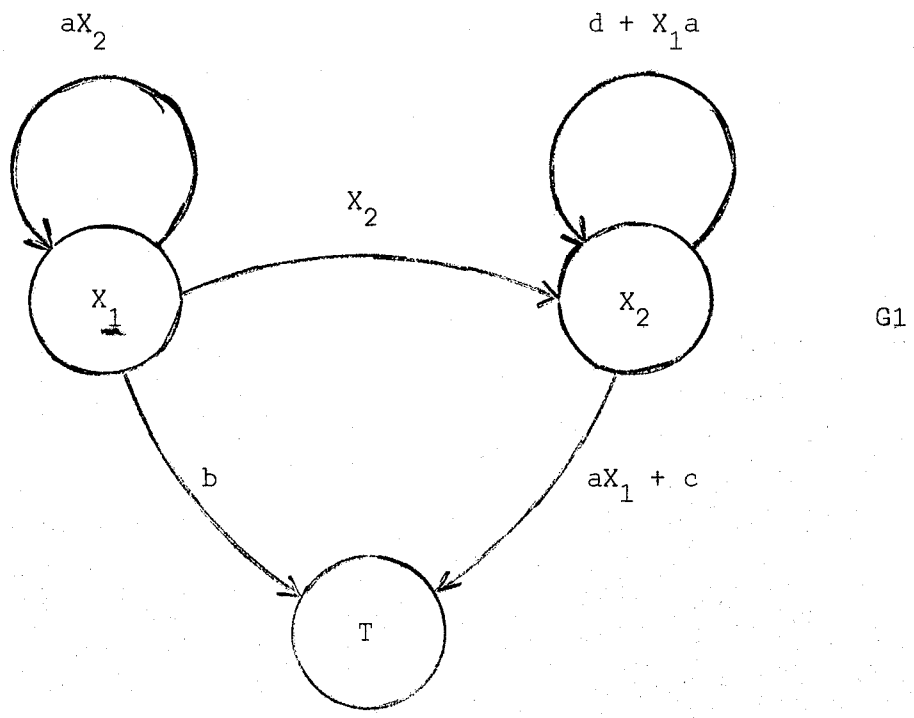
Réciproquement la donnée d'un tel graphe détermine de manière unique le système d'équations.

Avec la grammaire du paragraphe 1.4 on obtient les graphes de la Figure 2.

On peut construire à partir du système (S1) un système équivalent où toutes les chaînes figurant en partie droite ont leur premier symbole appartenant à V_T (Forme Normale de Greibach [Gre2] et, à partir de (S2), un autre où toutes les chaînes ont leur dernier symbole terminal.

---

(1) On suppose qu'il n'y a pas de règles du type (i) ou (ii) dans § 2.2.1, 2.2.2, 2.2.3.



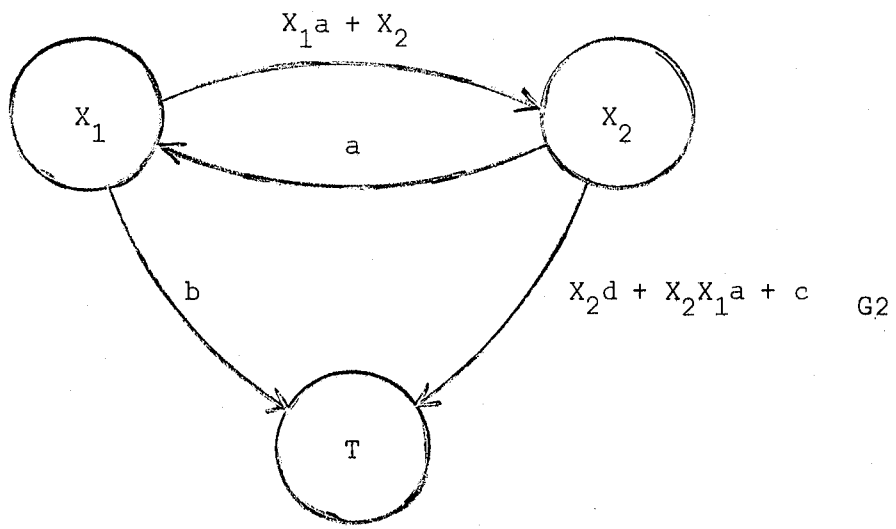


FIGURE. 2

De plus, si on applique successivement ces deux constructions, on obtient une forme normale où chaque chaîne en partie droite a un premier et un dernier symboles terminaux. [Ros1].

Nous allons appliquer la même méthode pour l'élimination de la récursivité à gauche.

2.2.2. Elimination de la récursivité à gauche

Soit H l'ensemble des symboles non-terminaux récursifs à gauche :

$$H = \{X_j \ : \ X_j \ell^+ X_j\}$$

Supposons H non vide et $H = \{X_1, \ldots, X_m\}$ $m \leqslant n$
Considérons le système suivant :

$$\text{(S3)} \qquad X_i = X_1\beta_{1i} + \ldots + X_m\beta_{mi} + \alpha_i \qquad 1 \leqslant i \leqslant m$$

α_i représente une somme de chaînes dont chaque premier symbole appartient à V - H.

Dans ce système, pour employer la terminologie algébrique, les occurrences des éléments de V_N - H ne sont plus considérées comme des variables, mais comme des constantes. Il correspond à (S3) un graphe G3, graphe relatif à la restriction de la relation ℓ à l'ensemble H.

Dans l'exemple précédent (S3) coïncide avec (S1) et G3 avec G1.
Réciproquement au graphe G3 correspond le seul système (S3).
Le système (S3) peut être écrit sous la forme matricielle :

$$\text{(S3)} \qquad X = XB + A$$

où X et A sont des matrices ligne d'éléments X_i et α_i
B est une matrice carrée m × m d'éléments β_{ij} ($i^{ème}$ ligne, $j^{ème}$ colonne).
La solution de (S3) est :

$$X = AB^*$$

où $B^* = I + B + B^2 + \ldots + B^q + \ldots$
I est la matrice m × m d'éléments $\delta_{ij} = \begin{matrix} \varepsilon & \text{si } i = j \\ \varphi & \text{si } i \neq j \end{matrix}$

Ce résultat peut se démontrer algébriquement [Sal] ou bien à l'aide du graphe associé [Ros1]. Précisons que l'élément β^*_{ij} de la matrice B^* est l'ensemble des séquences (obtenues par concaténation) d'étiquettes correspondant à tous les chemins du sommet X_j au sommet X_i dans le graphe.

La solution pour X_i est l'ensemble des séquences d'étiquettes correspondant à tous les chemins du sommet X_i au sommet T. B^* est donc une matrice dont les éléments sont des expressions régulières sur le vocabulaire total V.

On ne calcule pas cette matrice, mais on utilise l'identité :

$$B^* = I + BB^*$$

la solution s'écrit :

$$X = A + ABB^*$$

Si on pose $BB^* = Y$, la matrice Y satisfait à l'équation $Y = B + BY$ et réciproquement cette équation a pour solution unique $Y = BB^*$.

On obtient un système (S3') équivalent à (S3) :

$$\text{(S3')} \qquad \begin{aligned} X &= AY + A \\ Y &= BY + B \end{aligned}$$

La matrice Y est formée des éléments Y_{ij}, nouveaux symboles non-terminaux. Mais si dans le graphe G3, il n'existe aucun chemin du sommet j au sommet i, alors $\beta^*_{ij} = \delta_{ij}$. L'égalité $B^* = I + Y$ entraîne nécessairement $Y_{ij} = \varphi$. La réciproque est vraie. D'où $Y_{ij} = \varphi$ si et seulement si $X_j \ell^+ X_i$ est faux.

Règle de réduction

Dans le système (S3') on supprimera[1] un symbole non-terminal Y_{ij} si et seulement si $X_j \ell^+ X_i$ est faux.

Si cette règle n'était pas appliquée, dans la grammaire obtenue on aurait des symboles parasites.

---

(1) 'supprimer' veut dire : remplacer par φ dans le système (S3').

On pourrait montrer que la grammaire correspondant à (S3') n'a pas de récursivité à gauche. Mais ce système conduit à des règles qui ne peuvent satisfaire à la condition LL(1) n° 2. C'est pourquoi on pose $Z = Y + I$, ce qui donne (S3"), équivalent à (S3') :

(S3") $X = AZ$
$Z = I + BZ$

On introduit un nouveau symbole non-terminal Z_{ij} si $X_j \ell^+ X_i$ est vrai.

La <u>règle de réduction</u> devient :
Si $Y_{ij} = \varphi$ alors $Z_{ij} = \delta_{ij}$ d'où deux cas se présentent :

- $i = j$ alors $Z_{ii} = \varepsilon$ on remplace Z_{ii} par ε dans l'équation $X = AZ$, puis on supprime Z_{ii} dans l'équation $Z = I + BZ$
- $i \neq j$ alors $Z_{ij} = \varphi$ on supprime Z_{ij}.

Remarque

La relation ℓ^+ utilisée dans la règle de réduction concerne toujours le système (S3) et le graphe G3.

On a construit des grammaires correspondant à (S3') et (S3") qui sont équivalentes à la grammaire initiale. Il reste à montrer que, dans ces deux grammaires, aucun symbole non-terminal est récursif à gauche.

Au moins un α_i est différent de φ car alors on aurait des symboles parasites. La matrice A est formée des α_i qui ne peuvent pas dériver sur la chaîne vide ε (il n'y a pas de règle du type (i)) et dont le premier symbole à gauche ne peut être, par hypothèse, récursif à gauche. Donc X_i n'est pas récursif à gauche quel que soit i.

On ne peut pas avoir $\beta_{ij} \overset{*}{\rightarrow} \varepsilon$ (il n'y a pas de règle (i) ou (ii)), donc les symboles Y_{ij}, Z_{ij} ne sont pas récursifs à gauche.

Théorème

Pour toute grammaire CF ne possédant ni règles du type (i) ou (ii), ni symboles parasites ou inaccessibles, en appliquant la transformation du système (S3) en système (S3") et la règle de réduction, on peut construire une grammaire équivalente sans symbole non-terminal récursif à gauche.

Cette transformation ne conserve pas la relation ℓ. Cette transformation conserve la non-ambiguïté [Wo1].

2.2.3. Algorithme de transformation

Appliquons la transformation définie au paragraphe précédent à la grammaire :

$$X_1 = X_1 a X_2 + X_2 X_2 + b$$
$$X_2 = X_2 d + X_2 X_1 a + a X_1 + c$$

la forme matricielle du système d'équations (S3) s'écrit :

$$\| X_1 \quad X_2 \| = \| X_1 \quad X_2 \| \left\| \begin{matrix} aX_2 & \varphi \\ X_2 & d + X_1 a \end{matrix} \right\| + \| b \quad aX_1 + c \|$$

Compte tenu de la règle de réduction le système (S3") est :

$$\| X_1 \quad X_2 \| = \| b \quad aX_1 + c \| \left\| \begin{matrix} Z_{11} & \varphi \\ Z_{21} & Z_{22} \end{matrix} \right\|$$

$$\left\| \begin{matrix} Z_{11} & \varphi \\ Z_{21} & Z_{22} \end{matrix} \right\| = \left\| \begin{matrix} \varepsilon & \varphi \\ \varphi & \varepsilon \end{matrix} \right\| + \left\| \begin{matrix} aX_2 & \varphi \\ X_2 & d + X_1 a \end{matrix} \right\| \left\| \begin{matrix} Z_{11} & \varphi \\ Z_{21} & Z_{22} \end{matrix} \right\|$$

la grammaire transformée est :

$$X_1 = bZ_{11} + aX_1 Z_{21} + cZ_{21}$$
$$X_2 = aX_1 Z_{22} + cZ_{22}$$
$$Z_{11} = \varepsilon + aX_2 Z_{11}$$
$$Z_{21} = X_2 Z_{11} + dZ_{21} + X_1 a Z_{21}$$
$$Z_{22} = \varepsilon + dZ_{22} + X_1 a Z_{22}$$

Dans cet exemple deux simplifications sont possibles.

1°) On peut créer deux nouveaux symboles (non-) terminaux W_1 et W_2 tels que :

$$W_1 = aX_1 + c$$
$$W_2 = d + X_1 a$$

et la grammaire devient :

$$X_1 = bZ_{11} + W_1 Z_{21}$$
$$X_2 = W_1 Z_{22}$$
$$Z_{11} = \varepsilon + aX_2 Z_{11}$$
$$Z_{21} = X_2 Z_{11} + W_2 Z_{21}$$
$$Z_{22} = \varepsilon + W_2 Z_{22}$$

2°) On peut effectuer sur X_1, puis sur X_2, la transformation :

$$X_1 = W_3 Z_1 \qquad W_3 = X_2 X_2 + b$$
$$Z_1 = \varepsilon + aX_2 Z_1$$

$$X_2 = W_1 Z_1'$$
$$Z_1' = \varepsilon + W_2 Z_1'$$

On a économisé un symbole non-terminal et la récursivité à gauche est effectivement éliminée. On peut généraliser ceci en se ramenant à un problème de la théorie des graphes, qui est celui de la réduction d'un graphe fini.

Définition

On appelle <u>clique maximale</u> C_i un sous-ensemble de H réunissant tous les éléments satisfaisant à :

X_j, X_k appartiennent à C_i si et seulement si

$$X_j \ell^+ X_k \text{ et } X_k \ell^+ X_j.$$

Le sous-ensemble ainsi défini est une clique maximale par rapport au graphe de la relation ℓ^+ et une composante fortement connexe par rapport au graphe de la relation ℓ.

Tout élément de H appartient à une clique maximale et une seule parce que la relation de définition est une relation d'équivalence :

$$X_j eq X_k \text{ si et seulement si } X_j \ell^+ X_k \text{ et } X_k \ell^+ X_j$$

- réflexive $X_i \ell^+ X_i$ puisque X_i appartient à H
- symétrique $X_i \ell^+ X_j$ entraîne $X_j \ell^+ X_i$ et réciproquement
- transitive $X_i \ell^+ X_j$ et $X_j \ell^+ X_i$ entraîne $X_i \ell^+ X_k$ et $X_k \ell^+ X_i$
  $X_j \ell^+ X_k$ et $X_k \ell^+ X_j$

et que chaque clique maximale est une classe d'équivalence.

H étant ainsi partitionné, à chaque clique maximale on peut faire correspondre un système d'équations :

$$X_j = X_1 \beta_{1j} + \ldots + X_q \beta_{qj} + \alpha_j \qquad 1 \leq j \leq q$$

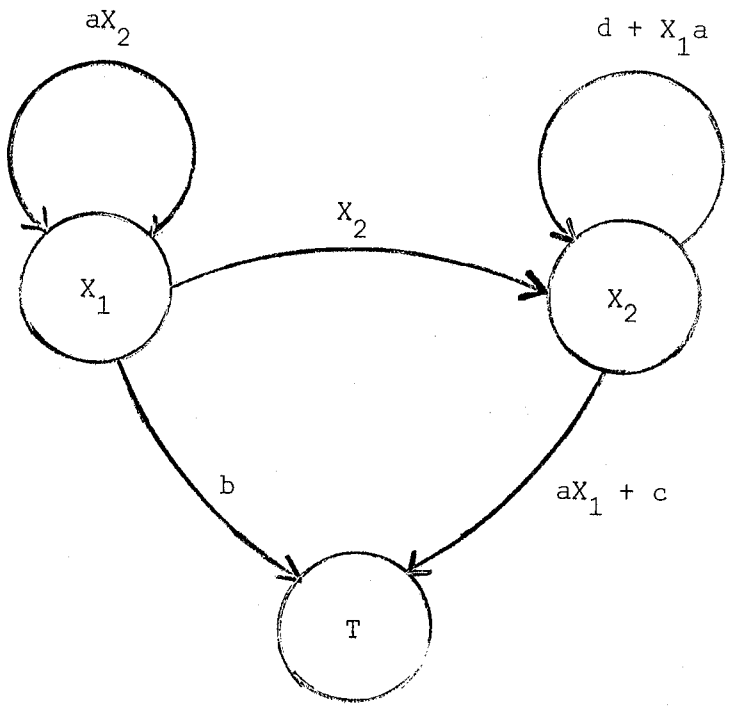
avec $C_i = \{X_1, \ldots, X_q\}$ et toute chaîne dans α_j a comme premier symbole un élément de $V - C_i$. (S4) est l'ensemble des systèmes associés à ces cliques. A (S4) il correspond un graphe G4, qui est unique puisque les cliques maximales sont disjointes deux à deux.

Dans l'exemple $\{X_1\}$ et $\{X_2\}$ sont deux cliques maximales et le système (S4) est composé de deux équations simples :

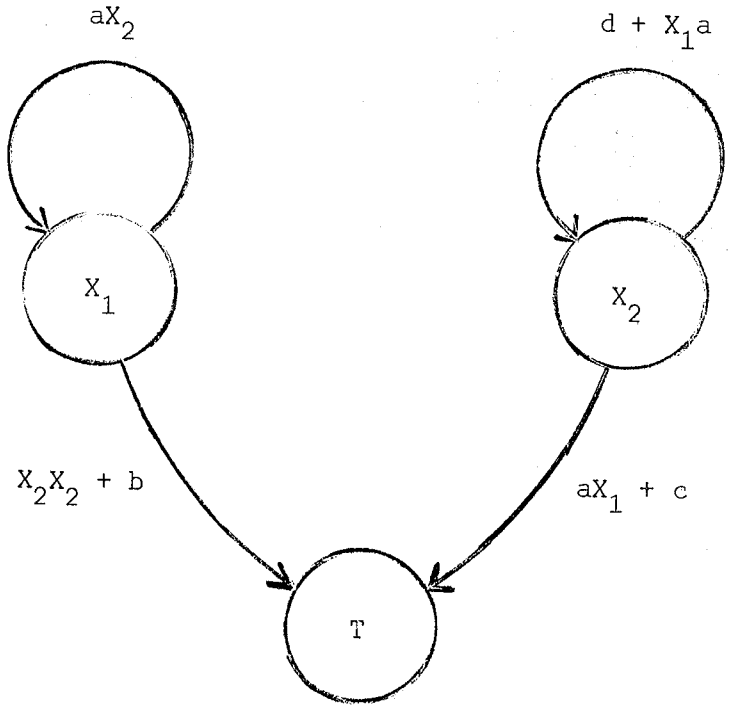
$$X_1 = X_1 a X_2 + X_2 X_2 + b$$
$$X_2 = X_2(d + X_1 a) + a X_1 + c$$

et le graphe G4 est différent du graphe G3 (Fig. 3).

Réciproquement le graphe G4 détermine le système S4.



G3



G4

FIGURE. 3

Comme dans le paragraphe 2.2.2 on peut démontrer qu'en transformant séparément chaque sous-système de S4 correspondant à chaque clique, on élimine la récursivité à gauche. Mais on n'a plus besoin de la règle de réduction. En effet chaque système transformé est tel que $Y_{ij} \neq \varphi$ et $Z_{ij} \neq \delta_{ij}$. Dans chaque clique maximale pour tous sommets X_j, X_i il existe un chemin de X_j à X_i.

Théorème

Pour éliminer la récursivité à gauche dans une grammaire CF, il est suffisant d'appliquer la transformation du paragraphe 2.2.2 successivement aux sous-ensembles de règles correspondant aux cliques maximales, sans 'règle de réduction'.

D'autre part, on peut créer un non-terminal W chaque fois que α_i ou β_{ij} dans le système étudié comporte plus d'une chaîne et créer la règle $W \to \alpha_i$ ou $W \to \beta_{ij}$. On remplace alors α_i ou β_{ij} par W dans le système d'équations.

Description de l'algorithme 2.2.3.

Compte tenu des simplifications, le programme effectivement utilisé fonctionne comme suit :

1°) Calcul de la relation ℓ à partir des règles de la grammaire donnée. Ici on peut se limiter à la restriction de ℓ à $V_N X V_N$. ℓ est représentée par une matrice booléenne.

2°) Calcul de la fermeture transitive ℓ^+, matrice carrée d'éléments PREMIER (i, j).

3°) Recherche des cliques maximales et transformation.

Pas 1 : initialement i = 1, l'ensemble E est égal à H.

Pas 2 : si X_i n'appartient pas à E alors changer i en i + 1 et reprendre le Pas 2.

Pas 3 : calcul de l'ensemble C_i

$$C_i = X_i \cup \{X_j : X_i \ell^+ X_j \text{ et } X_j \ell^+ X_i,\ j \neq i\}$$

en cherchant tous les X_j tels que :

PREMIER (i, j) = vrai et PREMIER (j, i) = vrai

Pas 4 : appel de la procédure de transformation avec, en paramètres, l'ensemble C_i et le nombre de ses éléments. La procédure traite les règles correspondant à C_i, calcule les matrices A et B, crée les nouveaux symboles et les nouvelles règles.

Pas 5 : l'ensemble E - C_i remplace E et on recommence au Pas 2 tant que E n'est pas vide.

Remarque

Une modification a été apportée au programme (§ 6.1).

2.2.4. Cas des grammaires possédant des règles du type (i) ou (ii)

Une grammaire CF peut toujours être réduite en une grammaire ne possédant pas de règles du type (i) ou (ii). L'élimination de la récursivité à gauche est donc toujours possible. Mais il est intéressant d'étudier le cas des grammaires possédant de telles règles.

La transformation décrite dans les paragraphes précédents peut ne pas éliminer toutes les récursivités à gauche.

Exemple 1

La grammaire ambiguë :

$$X = X + a$$

donne :

$$X = aZ$$
$$Z = \varepsilon + Z$$

Z est récursif à gauche.

Exemple 2

La grammaire :

$$X_1 = \varepsilon + a + X_1X_2b$$
$$X_2 = X_1c$$

est transformée en :

$$X_1 = aZ_{11} + Z_{11}$$
$$X_2 = aZ_{12} + Z_{12}$$
$$Z_{11} = \varepsilon + X_2bZ_{11} + cZ_{21}$$
$$Z_{12} = X_2bZ_{12} + cZ_{22}$$
$$Z_{21} = \varphi$$
$$Z_{22} = \varepsilon$$

X_2 et Z_{12} sont récursifs à gauche. La récursivité de X_2 dans la grammaire initiale était 'cachée'. Autre anomalie : Z_{21} et Z_{22} sont des symboles parasites, mais la règle de réduction ne s'applique pas puisque les relations $X_1 \ell X_2$ et $X_2 \ell X_2$ sont vérifiées.

Tout ceci s'explique par le fait que la transformation s'effectue par rapport à la relation de dépendance à gauche <u>apparente</u> ℓ_a (ou, ce qui revient au même par rapport au graphe G3) et non la relation ℓ (en l'absence de règle du type (i), ℓ_a est identique à ℓ). On va étudier les critères qui permettent de décider si on peut éliminer la récursivité à gauche à l'aide de l'algorithme § 2.2.3.

<u>Si $X\ell_aY$ alors $X\ell Y$.</u> Mais la réciproque n'est pas vraie en général. On en déduit que toute composante fortement connexe relative à ℓ_a est contenue dans (ou égale à) une composante fortement connexe relative à ℓ.

On peut généraliser l'algorithme § 2.2.3 :

Algorithme 2.2.4.

Pour chaque composante fortement connexe <u>relative à ℓ</u>, on effectue la transformation du paragraphe 2.2.2, avec la <u>règle de réduction</u> suivante :
créer Z_{ij} si et seulement si $X_j \ell_a^+ X_i$, sinon $Z_{ij} = \delta_{ij}$.

Considérons le système (S5) relatif à une composante fortement connexe C_i :

(S5) $X_j = \alpha_j + X_k \beta_{kj}$ (1)

pour tout j, k X_j, X_k appartiennent à C_i.

(S5') $X_j = \alpha_k Z_{kj}$ (1)

$Z_{ij} = \delta_{ij} + \beta_{ik} Z_{kj}$

(S5') est le système obtenu après transformation.

Supposons qu'il existe une suite d'indices $i_1, i_2, \ldots, i_p$ avec $p \geq 1$ tels que :

$$\beta_{i_1 i_2} \overset{*}{\rightarrow} \varepsilon, \ \beta_{i_2 i_3} \overset{*}{\rightarrow} \varepsilon, \ \ldots, \ \beta_{i_p i_1} \overset{*}{\rightarrow} \varepsilon$$

ceci entraîne dans le système (S5) :

$$X_{i_1} \overset{+}{\rightarrow} X_{i_p} \overset{+}{\rightarrow} \ldots X_{i_2} \overset{+}{\rightarrow} X_{i_1}$$

et dans le système (S5') :

$Z_{i_1 j} \overset{+}{\rightarrow} Z_{i_2 j}$ quelque soit j

$Z_{i_2 j} \overset{+}{\rightarrow} Z_{i_3 j}$ " " "

.....

$Z_{i_p j} \overset{+}{\rightarrow} Z_{i_1 j}$ quelque soit j

Donc, si on a des dérivations du type $X_i \overset{+}{\rightarrow} X_i$ dans (S5) on a obligatoirement des dérivations $Z_{ik} \overset{+}{\rightarrow} Z_{ik}$ dans (S5'). Il n'est pas possible d'éliminer ces récursivités en répétant la transformation sur (S5').

---

(1) On adopte la convention de somme sur les indices répétés.

Remarquons que, au point de vue algébrique, le système d'équations (S5) a dans ce cas une infinité de solutions [Sal], et la transformation n'est théoriquement plus applicable.

On va imposer à la grammaire initiale une condition moins forte, à savoir qu'il n'existe pas de dérivation :

$$X_{i_1} \overset{+}{\rightarrow} X_{i_2} \overset{+}{\rightarrow} \dots \overset{+}{\rightarrow} X_{i_p} \overset{+}{\rightarrow} X_{i_1}$$

Avec cette condition, on montrera qu'il est possible d'éliminer la récursivité à gauche.

Lemme

Dans le système (S5') le symbole non-terminal X_j est récursif à gauche si et seulement si X_j vérifie au moins l'une des propriétés (P1) ou (P2)

(P1) il existe k tel que $\alpha_k = X_{p_1} X_{p_2} \dots X_{p_r} X_j \psi$

- X_j appartient à C_i
- $X_{p_s} \overset{+}{\rightarrow} \varepsilon$ pour s = 1, 2, ..., r
- $r \geqslant 1$

(P2) il existe k tel que $\alpha_k \overset{*}{\rightarrow} \varepsilon$ et $m_1, m_2, \dots, m_t, m$ tel que

$$\beta_{m_1 m_2} \overset{*}{\rightarrow} \varepsilon,\ \beta_{m_2 m_3} \overset{*}{\rightarrow} \varepsilon,\ \dots,\ \beta_{m_{t-1} m_t} \overset{*}{\rightarrow} \varepsilon,$$

$$\beta_{m_t m} = X_{p_1} X_{p_2} \dots X_{p_r} X_j \psi$$

$$m_1 \neq m_2 \neq \dots \neq m_t \neq m,\ m_1 = k \text{ et } t \geqslant 1$$

- X_j appartient à C_i
- $X_{p_s} \overset{+}{\rightarrow} \varepsilon$ pour s = 1, 2, ..., r
- $r \geqslant 0$

Démonstration

- si X_j a la propriété (P1) alors $X_q \ell X_j$
- si X_j a la propriété (P2) alors

$$X_q \ell Z_{kq},\ Z_{kq} \ell Z_{m_2 q},\ \ldots,\ Z_{m_{t-1} q} \ell Z_{m_t q},\ Z_{m_t} \ell X_q$$

pour tout X_q appartenant à C_i
donc $X_j \ell^+ X_j$ est vrai et X_j est récursif à gauche.

Inversement supposons que X_j soit récursif à gauche. On a les possibilités suivantes :

1°) Il existe k tel que $\alpha_k \overset{+}{\to} X_j \psi$. Si X_j n'a pas la propriété (P1) alors

$$\alpha_k = X_{p_1} \ldots X_{p_r} X_q \psi \text{ et } X_q \ell^+ X_j \text{ est vrai}$$

X_q appartient nécessairement à C_i (puisque $X_j \ell X_q$ est vrai)

Il doit exister k' tel que

$$\alpha_{k'} \overset{*}{\to} \varepsilon \text{ et } X_q \overset{+}{\to} Z_{k'q} \overset{+}{\to} X_j \psi'$$

mais on aura aussi $X_j \overset{+}{\to} Z_{k'j} \overset{+}{\to} X_j \psi'$ et la récursivité de X_j obéit à la deuxième possibilité.

2°) Il existe k tel que $\alpha_k \overset{*}{\to} \varepsilon$ et $X_j \overset{+}{\to} Z_{kj} \overset{+}{\to} X_j \psi'$

On examine l'équation :

$$Z_{kj} = \delta_{kj} + \beta_{ki} Z_{ij}$$

il peut exister m_2 tel que $\beta_{km_2} = X_{p_1} \ldots X_{p_r} X_j \psi'$ et la propriété (P2) est vérifiée.

il peut exister m_2 tel que $\beta_{km_2} \overset{*}{\to} \varepsilon$

ceci entraîne $Z_{kj} \overset{+}{\to} Z_{m_2 j}$ et on itère en examinant l'équation $Z_{m_2 j}$.

Cette méthode de recherche aboutit toujours car il y a un nombre fini de Z_{sj} à examiner et on ne peut pas retrouver un Z_{sj} déjà examiné.

Si on retrouvait par exemple Z_{kj} on aurait

$$\beta_{km_2} \overset{*}{\to} \varepsilon,\ \beta_{m_2m_3} \overset{*}{\to} \varepsilon,\ \ldots,\ \beta_{m_{t-1}k} \overset{*}{\to} \varepsilon$$

ce qui est interdit.

La propriété (P2) est donc bien vérifiée.

<u>Les propriétés (P1) et (P2) permettent de rechercher les récursivités cachées</u>. Nous proposons maintenant un algorithme pour les éliminer.

Algorithme d'élimination des récursivités cachées

Pas 1. On exécute l'algorithme 2.2.4. Chaque système (S5) est ainsi transformé en système (S5').

Pas 2. Pour chaque système (S5') on recherche les récursivités cachées en testant (P1) et (P2) avec la restriction suivante :

$$\underline{X_{p_s} \text{ n'appartient pas à } C_i \text{ pour tout } s = 1, 2, \ldots, r}$$

Si on ne trouve pas de récursivités cachées alors on a terminé. Sinon on passe au Pas 3.

Pas 3. Dans chaque système (S5') pour tout X_j qui a été trouvé au cours du Pas 2 on effectue l'opération suivante :

<u>substituer à gauche</u> (voir § 2.4) successivement

les X_{p_i} <u>dans α_k dans l'équation X_j</u> si (P1) est vraie

ou <u>dans β_{m_tm} dans l'équation Z_{m_tj}</u> si (P2) est vraie

de manière à faire apparaître le terme $X_j\psi$.

On recommence ensuite au Pas 1.

Les exécutions du Pas 2 seront en nombre fini parce que, en partie droite des équations, il ne peut y avoir qu'un nombre fini de symboles X_j satisfaisant à (P1) ou (P2).

Il en est de même pour le Pas 3. La substitution des X_{p_i} ne peut pas entraîner la substitution d'un X_q appartenant à C_i (sinon X_{p_i} serait récursif à gauche et appartiendrait à C_i). Mais il se peut que X_{p_i} soit récursif à gauche sans appartenir à C_i. Dans ce cas les substitutions successives bouclent si et seulement si on substitue un symbole non-terminal X_q tel que $X_q \overset{+}{\to} X_q$ et $X_q \overset{+}{\to} \varepsilon$.

Si on substitue X_q dans $X_q X_{p_{i+1}} \ldots X_{p_r} X_j \psi$

$$\text{on obtient} \begin{cases} X_{p_{i+1}} \ldots X_{p_r} X_j \psi \\ \text{et} \quad X_q X_{p_{i+1}} \ldots X_{p_r} X_j \psi \end{cases}$$

il faudra substituer X_q indéfiniment.

Inversement si $X_q \overset{+}{\to} X_q \alpha$ (α ne pouvant conduire à ε)

$X_q \overset{+}{\to} \varepsilon$

les substitutions seront en nombre fini.

Or on a interdit les symboles X_q tels que $X_q \overset{+}{\to} X_q$.

En conclusion on peut dire que :

Théorème

Si une grammaire possédant des règles du type (i) : $X \to \varepsilon$ ou (ii) : $X \to Y$ et des symboles récursifs à gauche $X_1, X_2, \ldots, X_m$ n'a pas de dérivations du type :

(D) $\quad X_{i_1} \overset{+}{\to} X_{i_2} \overset{+}{\to} \ldots \overset{+}{\to} X_{i_1}$

alors il est possible d'éliminer toutes les récursivités à gauche à l'aide de l'algorithme fondé sur les propriétés (P1) et (P2).

Remarque

Pour éliminer les récursivités du type (D) il faut recourir à une autre méthode (par exemple en utilisant une méthode d'élimination des règles du type (ii) $X \to Y$).

2.2.5. Fonctions sémantiques

Pour l'étude approfondie de l'inclusion des fonctions sémantiques dans la syntaxe les références sont [Lew, p. 448; Gri2, p. 52; Fos2, chap. 6 et 7]. L'essentiel est que, la méthode d'analyse étant prédictive et déterministe, quand on a lu un symbole en avant, l'analyseur peut identifier complètement la règle à appliquer (on entend ici par 'règle' la forme $A \rightarrow \alpha$). On peut donc inclure des fonctions à n'importe quel endroit dans une règle appartenant à une grammaire LL(1).

Remarque

Au contraire avec une grammaire LR(1), une telle fonction ne peut être activée que lorsque la règle a été vérifiée, une fonction sémantique apparaîtra toujours à la fin d'une règle.

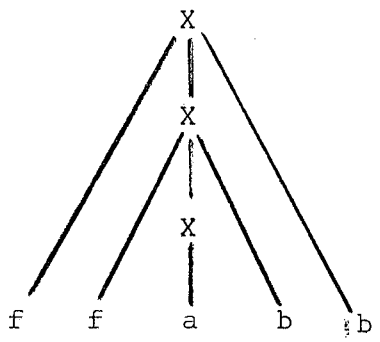
Conformément au transformateur de grammaire original, on offre la possibilité d'inclure des fonctions sémantiques dans la grammaire d'entrée, donc avant la transformation de cette grammaire. Mais cette dernière n'est pas forcément LL(1).

La transformation prend en charge ces noms de fonctions, lesquelles doivent être 'transformables'. Ceci veut dire que si l'activation d'une fonction sémantique se fait à un certain endroit dans le processus d'analyse, relativement à la grammaire initiale, l'activation de cette fonction doit se faire à l'endroit correspondant dans l'analyse relative à la grammaire transformée.

Soit f une fonction et la règle

$$X \rightarrow fXb \mid a$$

la fonction est activée avant chaque appel de X, ce qui peut être représenté par l'arbre :



Eliminons la récursivité à gauche (en l'absence de fonction sémantique) :

$$X \rightarrow aZ$$
$$Z \rightarrow \varepsilon \mid bZ$$

à quelque place que soit la fonction f dans ces règles, on ne peut réaliser l'appel récursif de f selon l'arbre ci-dessus.

Au contraire si une fonction sémantique apparaît à tout autre endroit, alors la transformation conserve l'activation correcte de cette fonction.

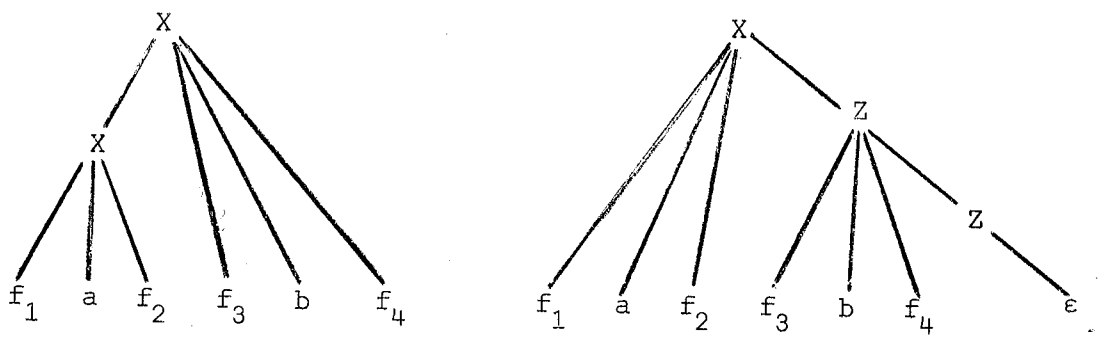
Soit

$$X \rightarrow Xf_3bf_4 \mid f_1af_2$$

on transforme les fonctions comme des symboles terminaux

$$X \rightarrow f_1af_2Z$$
$$Z \rightarrow \varepsilon \mid f_3bf_4Z$$

Relativement à chaque grammaire les arbres sont :



les fonctions apparaissent bien aux bonnes places.

La même argumentation est valable pour les récursivités d'ordre supérieur à 1. On déduit une règle générale :

Condition sur les fonctions sémantiques

Il est interdit de placer une fonction sémantique avant un symbole récursif à gauche dans une règle de la grammaire d'entrée.

Cette condition est vérifiée par le programme.

Si une fonction mal placée se présente, alors elle est ignorée (message d'erreur) et la transformation se poursuit.

Remarque

Cette condition n'est pas la seule. (voir § 2.3.4, 2.4.3, 2.5.4, 3.2.5).

2.3. Factorisation à gauche

2.3.1. Définition

Soit par exemple la règle :

$$X \rightarrow ABCD \mid ABE \mid AF \mid G$$

factoriser à gauche, c'est la transformer en :

$$X \rightarrow AY \mid G$$
$$Y \rightarrow BZ \mid F$$
$$Z \rightarrow CD \mid E$$

En notation ULD on écrirait :

$$X \rightarrow A < B < CD \mid E > \mid F > \mid G$$

La transformation est effectuée de la manière suivante.

Etant donnée une règle $X \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$, on calcule tous les plus grands facteurs gauches communs, soient

$$\alpha'_1, \alpha'_2, \ldots, \alpha'_p$$

et s'il n'en existe pas alors c'est terminé.

On forme la règle :

$$X \rightarrow \alpha'_1 X_1 \mid \alpha'_2 X_2 \mid \ldots \mid \alpha'_p X_p \mid \alpha_m \mid \ldots \mid \alpha_n$$

$X_1, X_2, \ldots, X_p$ étant des nouveaux symboles non-terminaux.

et les règles $X_1 \to \beta_{11} \mid \ldots \mid \beta_{1i}$

$\ldots$

$X_p \to \beta_{p1} \mid \ldots \mid \beta_{pq}$

Les chaînes β_{ij} sont telles que :

$$\alpha_1'\beta_{11} = \alpha_1,\ \alpha_1'\beta_{12} = \alpha_2,\ \ldots,\ \alpha_1'\beta_{1i} = \alpha_i,\ \alpha_2'\beta_{21} = \alpha_{i+1}$$

$$\ldots,\ \alpha_p'\beta_{p1} = \alpha_{m-q},\ \ldots,\ \alpha_p'\beta_{pq} = \alpha_{m-1}.$$

Sur les nouvelles règles $X_1, \ldots, X_p$ on recommence le processus de factorisation à gauche.

2.3.2.

Si on applique l'algorithme successivement à toutes les règles d'une grammaire CF, on obtient une <u>grammaire</u> nouvelle possédant la <u>propriété</u> suivante :
pour toute règle $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
on a :

$$t(\alpha_i) \neq t(\alpha_j) \text{ pour } i \neq j,\ \alpha_i \text{ et } \alpha_j \text{ différents de } \varepsilon.$$

<u>Remarque</u>

Si une règle du type $A \to \varepsilon \mid \varepsilon$ se présente (c'est une règle qui ne satisfait pas à la condition LL(1) n° 4), elle n'est pas transformée.

2.3.3. <u>Propriétés</u>

On étudie maintenant quelques propriétés intéressantes de la factorisation à gauche (on dira dans la suite simplement factorisation).

Désignons par V_N le vocabulaire non-terminal de la grammaire initiale, par V_N' l'ensemble des nouveaux symboles créés.

<u>La factorisation conserve la relation ℓ^+ restreinte au vocabulaire V_N.</u>

Pour le montrer, supposons qu'il existe B appartenant à V_N tel que

$$X\ell B \text{ dans la règle } X \to \alpha_1 \mid \ldots \mid \alpha_n$$

et regardons les règles obtenues après factorisation (voir § 2.3.1).

Si aucun $\alpha'_1, \ldots, \alpha'_p$ ne dérive sur la chaîne vide alors il existe au moins un α'_i, $i = 1, \ldots, p$ ou α_i, $i = m, \ldots, n$ tel que

$$\alpha'_i \text{ ou } \alpha_i = \alpha B \beta \qquad \text{avec } \alpha \overset{*}{\to} \varepsilon$$

et $X\ell B$ est vrai.

Si il existe α'_j tel que $\alpha'_j \overset{+}{\to} \varepsilon$ et si il existe β_{jk} tel que $\beta_{jk} \overset{*}{\to} B\psi$ dans la grammaire initiale alors dans la nouvelle grammaire $X \overset{+}{\to} X_j$ et il faut recommencer le raisonnement avec X_j et B.

On trouvera ainsi une suite finie $X \overset{+}{\to} X_j \overset{+}{\to} \ldots X_m \to \alpha B \beta$ avec $X\ell X_j, \ldots, X_m \ell B$, soit $X\ell^+ B$.

Remarque

Si la grammaire initiale n'a pas de règle du type $A \to \varepsilon$, alors c'est la restriction de la relation ℓ à $V_N \times V_n$ qui est conservée.

On peut maintenant préciser <u>les relations entre les éléments de V_N et les éléments de V'_N</u> puis celles <u>des éléments de V'_N entre eux</u>.

- $X\ell X_i$ est vrai si et seulement si $\alpha'_i \overset{+}{\to} \varepsilon$ (X_i apparaissant une seule fois en partie droite de règle, cela est évident) avec X appartient à V_N ou V'_N et X_i appartient à V'_N.

- $X_i \ell B$ est vrai si et seulement si $X_i \to \alpha B \beta$ avec X_i appartient à V'_N et B appartient à V_N et $\alpha \overset{*}{\to} \varepsilon$ (par définition de ℓ).

Il serait facile aussi de montrer que les restrictions de r^+ et s à $V_N \times V_N$ sont conservées par factorisation.

Remarques

1. Il est évident qu'une règle LL(1) ne peut pas être factorisée. Le fait de trouver un facteur gauche commun contredit la condition LL(1) n° 2.

Mais la réciproque n'est pas vraie : une règle qui ne peut pas être factorisée n'est pas, en général, LL(1).

Exemple

$$A \rightarrow Ab \mid a$$

2. L'élimination de la récursivité à gauche ne conserve pas la propriété 2.3.2.

Par exemple

$$X_1 \rightarrow X_1ab \mid X_2X_2 \mid b$$
$$X_2 \rightarrow X_1ac \mid X_2X_1a \mid c$$

donnera la règle :

$$Z_{11} \rightarrow \varepsilon \mid abZ_{11} \mid acZ_{21}$$

qui ne satisfait pas la propriété 2.3.2.

On devra donc éliminer la récursivité à gauche avant de faire les factorisations.

2.3.4. Cas des fonctions sémantiques

Si on inclut des fonctions sémantiques dans les règles de la grammaire originale, la transformation marche à une condition. Il faut que dans chaque plus grand commun facteur à gauche les occurrences de fonctions sémantiques soient identiques.

En effet les fonctions sémantiques sont transformées comme des symboles de base et il est facile de voir que si cette condition n'était pas réalisée alors la factorisation ne peut être faite correctement.

Exemple

$$X \rightarrow aF_1BC \mid aF_2BD \mid E$$

donne $X \rightarrow aY$

$Y \rightarrow F_1BC \mid F_2BD \mid E$

ce qui n'est pas conforme à l'algorithme 2.3.1, le facteur gauche devant être aB.

2.4. Substitution à gauche

2.4.1. Définition

Si étant données les règles :

$$A \to \alpha_1 \mid \ldots \mid \beta\alpha_j' \mid \ldots \mid \alpha_n$$

et

$$B \to \beta_1 \mid \ldots \mid \beta_n$$

on remplace la règle A par la règle :

$$A \to \alpha_1 \mid \ldots \mid \beta_1\alpha_j' \mid \ldots \mid \beta_m\alpha_j' \mid \ldots \mid \alpha_n,$$

on a effectué la substitution à gauche du non-terminal B dans la règle A. (on dira dans la suite substitution tout court, sauf indication contraire).

2.4.2. Propriétés

Il est bien connu que la substitution ne modifie pas le langage généré. Mais, à la différence de la factorisation (à gauche), nous avons la propriété suivante : Les relations ℓ^+, r^+, s ne sont pas conservées dans la substitution.

Démonstration

On voit que, dans la définition 2.4.1, $A\ell B$ n'est pas vrai dans la nouvelle règle A. Si α_j' conduit à la chaîne vide alors la relation ArB n'est pas vraie dans la nouvelle règle A. Enfin si α_j' commence par un non-terminal C alors BsC n'est plus vrai. Remarquons bien que ces relations pourraient être éventuellement encore vraies à cause d'autres règles de la grammaire.

Par exemple, on pourrait avoir un α_i tel que

$$\alpha_i \overset{+}{\to} B\alpha_i'$$

Il y a cependant un cas où les relations ne sont certainement pas conservées :

quand le symbole B de la définition 2.4.1 n'apparaît qu'une seule fois en partie droite dans toutes les règles de la grammaire, après substitution de B, B devient inaccessible.

Propriété importante

La substitution conserve la propriété LL(1).

Si on substitue une règle LL(1) dans une autre règle LL(1), la règle obtenue est encore LL(1). C'est la conséquence d'un théorème plus général sur les grammaires LL(k) [Kur2, p. 228], que nous transposons ici avec les notations définies au début :

Soit $B \to \alpha_1 \mid \ldots \mid \alpha_n$ une règle LL(k).

si une règle LL(k) $A \to \beta_1 \mid \ldots \mid xB\beta_j \mid \ldots \mid \beta_n$
est remplacée par :

$$A \to \beta_1 \mid \ldots \mid x\alpha_1\beta_j \mid \ldots \mid x\alpha_n\beta_j \mid \ldots \mid \beta_n$$

cette nouvelle règle est LL(k + $\ell(x)$). Le langage généré par la grammaire et les propriétés LL(k) possibles des autres règles ne sont pas affectées par cette modification.

Dans le cas que nous étudions ici, k = 1 et $\ell(x) = 0$.

2.4.3. Fonctions sémantiques

Si il y a des fonctions sémantiques, la substitution peut être toujours effectuée. Il est seulement nécessaire d'introduire une petite modification dans le programme, un (ou plusieurs) nom(s) de fonction pouvant apparaître à gauche du symbole non-terminal que l'on veut substituer.

Exemple

$$X \to fY \mid a$$

et

$$Y \to b \mid c$$

donnera :

$$X \to fb \mid fc \mid a$$

2.5. Factorisation à droite

2.5.1. Définition

Comme exemple, la règle :

$$X \rightarrow ACD \mid BCD \mid ED \mid F$$

est transformée en :

$$X \rightarrow YD \mid F$$
$$Y \rightarrow ZC \mid E$$
$$Z \rightarrow A \mid B$$

En notation ULD, on écrirait :

$$X \rightarrow < < A \mid B > C \mid E > D \mid F$$

C'est l'opération symétrique de la factorisation à gauche. Ici on factorise le plus grand commun facteur à droite.

2.5.2.

En factorisant à droite toutes les règles d'une grammaire CF on obtient une grammaire où, dans chaque règle, les derniers symboles de chaque partie droite sont différents deux à deux. De plus, si on a déjà effectué les factorisations à gauche, on obtient une grammaire, où chaque règle satisfait à la propriété 2.3.2 et à celle ci-dessus.

2.5.3. Propriétés

La factorisation à droite ne conserve pas en général la relation r, mais les restrictions de r^+, ℓ^+, s au vocabulaire initial V_N sont conservées.

Pour montrer cette propriété, il suffit d'utiliser des démonstrations symétriques de celles du paragraphe 2.3.3 (factorisation à gauche).

Si une règle est LL(1), la factorisation à droite de cette règle ne change pas la propriété LL(1).

Il n'est pas difficile de montrer que si l'une des règles obtenues après factorisation à droite n'est pas LL(1), alors la règle initiale n'est pas LL(1). Prenons par exemple dans § 2.5.1 la règle :

$Y \rightarrow ZC \mid E$

Supposons que la deuxième condition LL(1) ne soit pas vérifiée, soit Premier (ZC) et premier (E) non disjoints, on a :

Premier (ZC) = premier (A) $\cup$ premier (B)

$\cup$ si $Z \overset{+}{\rightarrow} \varepsilon$ alors premier (C)

d'où premier (A) et premier (E) non disjoints

ou premier (B) et premier (E) " "

ou premier (C) et premier (E) non disjoints.

La règle X initiale ne serait pas LL(1).

L'examen des autres conditions LL(1) conduit à la même conclusion.

On déduit de la propriété démontrée celle qui suit :

Pour toute grammaire LL(1), il existe une grammaire LL(1) factorisée à droite équivalente.

2.5.4. Fonctions sémantiques

Remarque symétrique de celle du paragraphe 2.3.4 en changeant 'gauche' en 'droite'.

CHAPITRE III

METHODES DE TRANSFORMATION

CHAPITRE III

METHODES DE TRANSFORMATION

3. METHODES DE TRANSFORMATIONS D'UNE GRAMMAIRE

3.1. On dispose de deux résultats théoriques intéressants.
D'abord toute grammaire LL(1) a une grammaire équivalente avec des règles de la forme :

$$A \to a_1\beta_1 \mid \dots \mid a_n\beta_n$$

ou

$$A \to a_1\beta_1 \mid \dots \mid a_n\beta_n \mid \varepsilon$$

où les a_i sont différents deux à deux et aucun a_i n'appartient à suivant (A) pour la règle de la 2ème forme. |Kn2|. Une méthode de transformation serait d'obtenir une grammaire sous forme normale de Greibach, puis d'essayer, par factorisations et substitutions, de se ramener à cette forme de grammaire LL(1). Mais cette grammaire aurait alors un trop grand nombre de règles pour être utilisable pour l'analyse syntaxique.

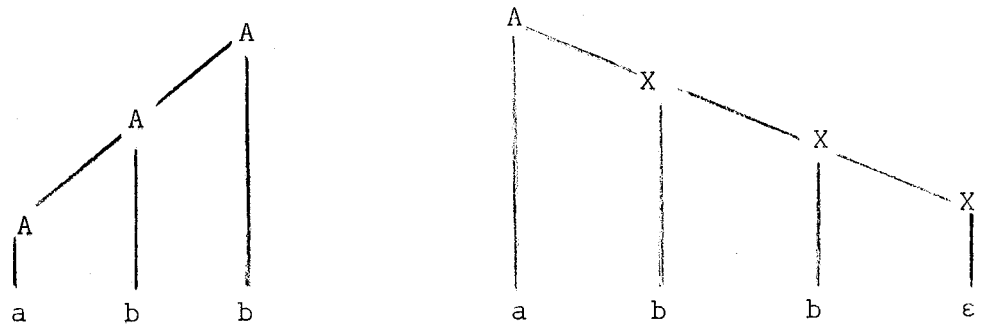
D'autre part, étant donnée une grammaire CF quelconque, on peut décider s'il existe une grammaire LL(1) sans règle du type $A \to \varepsilon$, qui est structurellement équivalente à l'originale [Pau].

Malheureusement l'équivalence structurelle est trop restrictive : on ne peut pas éliminer la récursivité à gauche, on ne peut pas en général factoriser.

Par exemple, aux règles :

1°) $A \to a \mid Ab$ ou 2°) $A \to aX$
$X \to \varepsilon \mid bX$

correspondent des structures différentes pour la chaîne abb.



On sait qu'une grammaire LL(1) sans règle $A \to \varepsilon$ génère un S-langage [Kur2], ce qui limite encore le résultat. De l'étude de ces deux résultats théoriques, on voit se dégager deux principes importants.

Le premier est de minimiser le nombre des transformations utilisées, de manière à obtenir une grammaire avec le moins de règles possibles. En effet, si la grammaire obtenue est effectivement LL(1), l'analyseur généré réalisera une économie de place et une économie de temps à l'exécution.

Le second est d'avoir une méthode la plus générale possible. C'est pourquoi on choisit des transformations avec équivalence faible, c'est-à-dire qu'on impose seulement que le langage généré soit le même. On perd alors la structure initiale du langage, mais ce n'est pas un inconvénient. Car on peut introduire dans les règles des actions ou fonctions sémantiques qui peuvent reconstituer la structure.

Exemple

Prenons les grammaires 1°) et 2°). Dans la deuxième nous insérons les fonctions f_1 et f_2 :

$$A \to af_1X$$
$$X \to \varepsilon \mid bf_2X$$

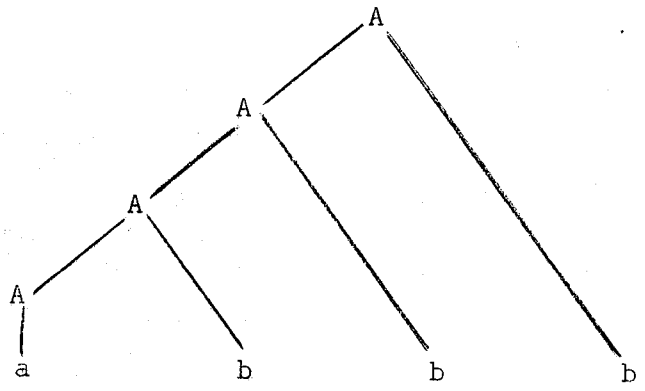
En langage de listes, les fonctions f_1 et f_2 réalisent les affectations suivantes :

f_1 : clist = cons (a, nil)
f_2 : clist = cons (clist, cons (b, nil))

L'analyse de la chaîne abbb génère la liste :

clist = ((((a)b)b)b)

équivalente à la structure :



qui correspond à la grammaire 1°).

Remarque

Dans les compilateurs comportant un seul passage, en général, la structure syntaxique n'est pas 'générée' à proprement parler, les fonctions sémantiques étant exécutées en même temps que l'analyse pour faire la compilation.

Dans ce qui suit, on va étudier la méthode de Foster qui reste actuellement la meilleure, puis indiquer quelques améliorations. On exposera brièvement la méthode de Paull et Unger qui est intéressante par elle-même. On verra aussi d'autres transformations particulières, utilisables dans certains cas.

3.2. Méthode de substitution-factorisation

Cette méthode, inventée par Foster [Fos] et appellée S.I.D. (Syntax Improving Device) se décompose en plusieurs étapes. On supposera qu'on a une grammaire initiale sans parasite et sans inaccessible [Mar]. Avant d'exposer cette méthode, on rappelle un algorithme important.

3.2.1. Algorithme de numérotation

Une propriété importante des grammaires sans symbole récursif à gauche est qu'on peut ordonner les symboles non-terminaux de telle manière que :

$X_j \ell^+ X_i$ entraîne $j > i$

Le graphe de la relation ℓ est un graphe orienté sans circuit. Il est bien connu que les sommets d'un tel graphe peuvent être ordonnés de telle manière que si il existe un chemin du sommet X_j au sommet X_i alors $j > i$.

L'algorithme est :

- le sommet 'terminal' T a le numéro zéro.
- les sommets $X_1, \ldots, X_i$ ayant été numérotés 1, 2, ..., i, on donne le numéro i + 1 à un sommet non encore numéroté et relié par un arc à un sommet déjà numéroté. Cet algorithme est simple à exécuter à l'aide de la matrice booléenne de la fermeture transitive ℓ^+.

Remarque

Cette numérotation permettrait aussi d'obtenir une matrice réduite pour ℓ^+ [Abr].

3.2.2. Elimination de la récursivité à gauche

Soit G0 la grammaire originale. La première étape est l'élimination de la récursivité à gauche, selon la méthode décrite au paragraphe 2.2.3. On obtient une grammaire G1, qui peut contenir des règles du type $A \to \varepsilon$. On a vu en 2.2.4 qu'il y avait des cas où certaines récursivités à gauche subsistaient. Ce fait sera vérifié dans l'étape suivante.

3.2.3. Première factorisation et calcul de ℓ^+

Toutes les règles de la grammaire G1 sont factorisées de manière à obtenir une grammaire G2 satisfaisant à la propriété 2.3.2.

$A \to \alpha_1 | \alpha_2 | \ldots | \alpha_n$

$l(\alpha_1) \neq l(\alpha_2)$

On calcule l'ensemble :

$$E = \{A_i : A_i \overset{+}{\rightarrow} \varepsilon\}$$

selon l'algorithme décrit dans [Gri2].

Remarque

Cet algorithme ne décide pas pour les règles du genre $A_i \rightarrow A_j \rightarrow \ldots \rightarrow A_i$

Mais elles font partie des règles interdites signalées au paragraphe 2.2.4.

L'ensemble E permet de calculer la relation ℓ en lisant les règles de la grammaire G2. A partir de ℓ on calcule ℓ^+. On regarde s'il existe des symboles A_i tels que $A_i \ell^+ A_i$. Si oui on est dans les cas cités au paragraphe 2.2.4 et on ne continue pas. Sinon, G2 satisfait à la condition LL(1) n° 1.

3.2.4. Substitution-factorisation

Pas 1. On numérote les symboles non-terminaux de la grammaire G2 suivant l'algorithme 3.2.1.

Pas 2. Pour chaque règle de G2 :

$$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$$

on calcule les ensembles Premier (α_i) et Premier (α_j) pour tout i, j tels que $i \neq j$, $\alpha_i \neq \varepsilon$, $\alpha_j \neq \varepsilon$

Si

Premier (α_i) et Premier (α_j) ne sont pas disjoints alors les cas suivants peuvent se présenter :

cas 1. $t(\alpha_i)$ est un terminal et $t(\alpha_j)$ est non-terminal.

cas 2. $t(\alpha_j)$ " " " et $t(\alpha_i)$ " " " .

cas 3. $t(\alpha_i)$ et $t(\alpha_j)$ sont non-terminaux.

(mais $t(\alpha_i)$ et $t(\alpha_j)$ ne peuvent pas être tous deux terminaux à cause de la propriété 2.3.2).

On forme, une fois que tous les couples (α_i, α_j) ont été examinés, l'ensemble des symboles non-terminaux se trouvant dans l'un de ces trois cas. On choisit le symbole B qui a le plus grand numéro (selon l'algorithme 3.2.1).

On substitue (§ 2.4) ce symbole dans la règle A. Cette substitution est unique parce que G2 satisfait à la propriété 2.3.2. (c'est-à-dire on substitue une seule fois B dans la règle A).

Ceci permet de réduire le nombre de transformations successives à effectuer (voir l'exemple ci-dessous).

Si une substitution a été faite dans la règle courante, on factorise. Sinon la règle satisfait à la condition LL(1) n° 2.

Une fois que toutes les règles de G2 ont été examinées, on a une nouvelle grammaire G3.

Si toutes les règles de G2 satisfaisaient à la condition LL(1) n° 2 (auquel cas G3 est identique à G2) alors on termine.

Sinon G3 satisfait encore à la propriété 2.3.2 et on effectue le Pas 3.

Pas 3. On calcule la relation ℓ^+ relative à la nouvelle grammaire G3. On reprend alors le Pas 1 où G3 remplace G2.

Remarque

A chaque itération du Pas 2 on ne testera pas bien sûr les règles déjà reconnues comme satisfaisant à la condition LL(1) n° 2.

Exemple

La grammaire :

$$A \to Bc \mid De$$
$$B \to Df \mid b$$
$$D \to d$$

les numéros des symboles :

D a le numéro 1
B a le numéro 2
A a le numéro 3

la substitution du non-terminal qui a le plus grand numéro donne :

A → Dfc | De | bc
B → Df | b
D → d

et en factorisant :

A → DX | bc
X → fc | e
B → Df | b
D → d

qui est LL(1).

Tandis que, en substituant le plus petit numéro :

A → Bc | de
B → Df | b
D → d

substitution : A → Dfc | bc | de
B → Df | b
D → d

substitution : A → dfc | bc | de
B → Df | b
D → d

factorisation : A → dX | bc
X → fc | e
B → Df | b
D → d

ici, deux substitutions de plus.

3.2.5. Fonctions sémantiques

Si il y a des fonctions dans la grammaire, les substitutions doivent s'effectuer comme en 2.4.3. Mais à chaque factorisation ces fonctions doivent satisfaire à la condition 2.3.4.

Le test de cette condition alourdirait le programme et chaque fois qu'elle ne serait pas vérifiée, le déroulement de l'algorithme de substitution-factorisation serait perturbé.

Il est préférable d'utiliser la méthode suivante :
on soumet au programme une grammaire G sans fonction.

Supposons qu'on ait obtenu une grammaire G' qui est LL(1). Alors on introduit les fonctions dans la grammaire G ce qui donne une grammaire G_f. Si la grammaire G'_f obtenue après transformations est LL(1), c'est que la condition 2.3.4 sur les fonctions sémantiques est bien vérifiée, sinon les règles non LL(1) sont indiquées et il est facile de trouver les fonctions ne vérifiant pas cette condition.

3.2.6. Remarque

Dans les calculs du Pas 3 de l'algorithme 3.2.4, il n'est pas nécessaire de calculer à nouveau l'ensemble E (voir § 3.2.3). En effet si un $A_i \overset{+}{\rightarrow} \varepsilon$, après substitution et factorisation ceci est toujours vrai. Il suffit donc d'ajouter à E les symboles non-terminaux nouvellement créés qui conduisent à la chaîne vide.

Par contre la relation ℓ^+ doit être recalculée entièrement à cause de la propriété de non-conservation dans la substitution (§ 2.4.2). Nous ne pouvons pas utiliser les propriétés de conservation de la factorisation (§ 2.3.3). Ce pas de l'algorithme sera donc coûteux en temps.

Une autre conséquence est le calcul, à chaque itération, de la numérotation (Pas 1 du paragraphe 3.2.4).

3.2.7. Etude de la méthode SID

On peut se demander pourquoi on élimine la récursivité à gauche avant de faire les substitutions et factorisations, pour la raison que les conditions LL(1) numéros 2, 3, 4 impliquent la condition n° 1.

D'une part, cela est nécessaire parce que, avec une récursivité à gauche, les substitutions-factorisations bouclent indéfiniment.

D'autre part, on a vu que l'élimination de la récursivité à gauche faisait appel, dans son principe, à l'opération étoile, tandis qu'un nombre fini de substitutions et de factorisations ne peut pas réaliser une telle opération.

Une autre limitation de SID est qu'elle ne permet pas de satisfaire aux conditions LL(1) n°s 3 et 4. Ce qui ne veut pas dire pour autant qu'il n'existe pas de grammaire LL(1) équivalente.

Exemple

word → number space space
number → number digit | number space digit | digit

space et digit sont des symboles terminaux
number est récursif à gauche d'où :

word → number space space
number → digit numbertail
numbertail → ε | digit numbertail | space digit numbertail

cette grammaire est LL(2) mais n'est visiblement pas LL(1) à cause de la condition 3. La règle numbertail n'est pas transformée par SID. Or il existe une grammaire LL(1) équivalente :

word → number space
number → digit numbertail
numbertail → digit numbertail | space spacetail
spacetail → ε | digit numbertail

Il peut être intéressant de se limiter volontairement à un certain nombre de substitutions et de factorisations, c'est-à-dire à un nombre d'itérations dans l'algorithme § 3.2.4. Ce nombre sera fourni au programme comme paramètre (voir § 6.2).

La substitution peut donner des symboles inaccessibles. Il faut donc les éliminer après l'exécution de SID.

SID peut boucler indéfiniment.

Nous avons trouvé que SID échoue dans les cas où il existe A, B, C tel que :

(1) $A \overset{+}{\rightarrow} \omega B \psi_1$ | $B \overset{+}{\rightarrow} \omega B \psi_2$ (éventuellement $\omega \overset{*}{\rightarrow} \varepsilon$)

et

(2) $B \overset{+}{\rightarrow} \alpha B \beta$
$C \overset{+}{\rightarrow} \alpha C \gamma$

Démonstration

L'hypothèse (1) entraîne qu'on a

(3) $\alpha_i \overset{*}{\rightarrow} \omega B \psi_1$
$\alpha_j \overset{*}{\rightarrow} \omega C \psi_2$

dans la règle :

$$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$$

(éventuellement α_i peut être égal à α_j, ou bien il peut exister plusieurs α_i et/ou plusieurs α_j satisfaisant à la propriété (3)).

Par substitutions et factorisations successives, on arrivera obligatoirement à une règle du type

$$X \rightarrow \alpha' B \beta \psi_1 \mid \alpha' C \gamma \psi_2$$

où α' est un facteur à droite de α, ou bien ε, ou bien α (en effet il est possible que α ait été découpé en morceaux par une ou des factorisations précédentes).

On effectue alors la factorisation suivante :

(4) $X \rightarrow \alpha' Y$
$Y \rightarrow B \beta \psi_1 \mid C \gamma \psi_2$

on se retrouve dans une configuration identique à (1) et les relations (2) sont toujours vérifiées. Nous sommes ramenés au problème initial et l'algorithme boucle.

Comment faire pour éviter une telle boucle ? Le test des conditions (1) et (2) serait très long et compliqué à réaliser. Nous avons pensé au test suivant :
si l'on trouve 2 fois la configuration (4) au cours des substitutions et factorisations à partir d'un symbole non-terminal A, alors arrêter les transformations. Malheureusement on peut retrouver la configuration (4) un nombre fini de fois sans que pour cela SID boucle. L'exemple qui suit le montre :

La grammaire :

$$A \to \alpha BBE \mid \alpha CCD$$
$$B \to \alpha$$
$$C \to \alpha$$

est transformée successivement en :

1°)

configuration (4) $\begin{cases} A \to \alpha X \\ X \to BBE \mid CCD \end{cases}$ substitution $X \to \alpha BE \mid \alpha CD$

$B \to \alpha$

$C \to \alpha$

2°)

$A \to \alpha X$

configuration (4) $\begin{cases} X \to \alpha Y \\ Y \to BE \mid CD \end{cases}$ substitution $Y \to \alpha E \mid \alpha D$

$B \to \alpha$

$C \to \alpha$

3°)

$A \rightarrow \alpha X$

$X \rightarrow \alpha Y$

$Y \rightarrow \alpha Z$

$Z \rightarrow E \mid D$

Pour chaque grammaire il serait sans doute possible de trouver une borne supérieure pour le nombre de répétitions de la configuration (4) dues à de telles règles. Au-delà de cette borne on est alors sûr que SID boucle. Mais ce test ne serait pas intéressant du point de vue pratique. Car on risque d'attendre assez longtemps avant d'arrêter l'algorithme.

Un test serait donc très lourd à réaliser. On peut cependant se contenter de vérifier les conditions (1) et (2) dans les cas particuliers du type :

$A \rightarrow B\psi_1 \mid C\psi_2$

$B \rightarrow aB\beta$

$C \rightarrow aC\gamma$

On les rencontre souvent dans les grammaires des langages de programmation.

<u>Réciproquement les conditions (1) et (2) sont nécessaires pour que SID boucle</u>.

<u>Démonstration</u>

S'il n'y a pas de symbole non-terminal récursif dans la grammaire, le nombre de substitutions et de factorisations successives possibles est fini. Donc si SID boucle ce ne peut être qu'à cause de règles récursives. Les récursivités à gauche ayant été éliminées, il y a seulement des récursivités du type de celles de la condition (2) où α ne peut pas être égal à la chaîne vide. Si (1) n'est pas vrai il n'y a aucune raison de substituer B ou C au cours de l'algorithme de substitutions-factorisations, dans les règles obtenues à partir du non-terminal A. Supposons donc que la condition (1) est vérifiée.

Soit $B \overset{+}{\rightarrow} \alpha B \beta$ et $C \overset{+}{\rightarrow} \alpha\gamma$

Les substitutions et factorisations successives seront en nombre fini pour C, puisque C n'est pas récursif. C'est-à-dire, quand α aura été factorisé, SID s'arrêtera.

Remarque

Si $C \overset{+}{\rightarrow} B$ on est ramené au même problème. La condition (2) est donc bien nécessaire.

Exemple

$S \rightarrow B \mid C$
$B \rightarrow aB \mid b$
$C \rightarrow aC \mid c$

cette grammaire génère le langage $\{a^n b\} \cup \{a^n c\}$ et satisfait aux conditions (1) et (2). SID bouclera pour cette grammaire.

Il se peut qu'il existe quand même une grammaire LL(1) équivalente. Dans l'exemple :

$S \rightarrow aS \mid b \mid c$

Il se peut aussi qu'il n'en existe pas :

$S \rightarrow aS \mid aTb \mid a$
$T \rightarrow aTb \mid \varepsilon$

cette grammaire génère le langage $\{a^n b^m : n \geqslant m\}$ qui n'est LL(k) pour aucun k [Ros2].

3.3. Méthode de Paull et Unger

Paull et Unger présentent un algorithme pour obtenir, si elle existe, une grammaire LL(1) structurellement équivalente et sans règle $A \rightarrow \varepsilon$ à partir d'une grammaire CF quelconque.

La méthode consiste en les opérations suivantes : Soit G0 la grammaire originale

- construire une grammaire G1 qui n'a pas deux règles qui ont la même partie droite (G1 structurellement équivalente à G0)
- transformer G1 en une grammaire G2 qui n'a pas à la fois deux règles du type $X_i \to X_j\alpha$ et $X_i \to X_k\alpha$ et qui est structurellement équivalente à G1
- si G2 est LL(1), c'est la grammaire cherchée, sinon il n'en existe pas.

On a déjà dit que cette méthode est trop restrictive (elle ne permet pas de faire toutes les factorisations, d'éliminer la récursivité à gauche).

C'est la deuxième étape qui est intéressante et nous allons simplement la développer sur l'exemple des auteurs [Pau].

G0 :
$X_o \to X_1X_1 \mid aX_2 \mid aX_4$
$X_1 \to X_3b \mid X_5b \mid cX_3 \mid hX_5$
$X_2 \to dX_4 \mid e$
$X_3 \to bc \mid X_2X_1 \mid X_4X_1$
$X_4 \to dX_2 \mid fX_1a$
$X_5 \to g \mid X_2X_1 \mid X_4X_1$

G1 :
$X_o \to X_1X_1 \mid aX_2 \mid aX_4$
$X_1 \to X_{31}b \mid X_{32}b \mid X_{33}b \mid cX_{32} \mid cX_{31} \mid hX_{31} \mid hX_{33}$
$X_2 \to e \mid dX_4$
$X_4 \to dX_2 \mid fX_1a$
$X_{31} \to X_2X_1 \mid X_4X_1$
$X_{32} \to bc$
$X_{33} \to g$

Le principe de la méthode est de réduire dans chaque règle les termes figurant en partie droite qui ont la même <u>forme</u>.

La <u>forme</u> d'une chaîne $A_1A_2 \ldots A_n$ appartenant à V^* est définie par l'application f :

$$A_1A_2 \ldots A_n \overset{f}{\rightarrow} B_1B_2 \ldots B_n$$

si A_i appartient à V_T alors $B_i = A_i$

si A_i " " V_N alors $B_i = F$

F étant un symbole n'appartenant pas à V_N.

Par exemple la <u>forme</u> de aA_1bA_2c est aFbFc.

On utilise aussi des <u>équations structurelles</u> :

soient α et β des chaînes appartenant à V_N^*,

$\alpha = \beta$ si et seulement si α et β génèrent les mêmes chaînes terminales avec les mêmes structures.

On commence avec l'équation (structurelle) simple

(1) $Y_o = X_o$

et la règle $X_o \rightarrow X_1X_1$.

(les symboles Y désignent les non-terminaux de G2) Y_1 et Y_2 étant des nouveaux symboles non-terminaux on forme l'équation $Y_1Y_2 = X_1X_1$ qui conduit à l'équation simple :

(2) $Y_1 = Y_2 = X_1$

et la règle $Y_o \rightarrow Y_1Y_2$

Ensuite à partir de $X_o \rightarrow aX_2 \mid aX_4$ on forme la règle :

$$Y_o \rightarrow aY_3$$

et l'équation :

(3) $Y_3 = X_2 + X_4$

(cette transformation est analogue à la factorisation à gauche).

On obtient finalement pour Y_o (compte tenu de (2))

$$\underline{Y_o \rightarrow Y_1Y_1 \mid aY_3}$$

La règle $X_1 \to X_{31}b \mid X_{32}b \mid X_{33}b$ conduit à :

$$Y_1 \to Y_4 b$$

et à l'équation :

$$Y_4 = X_{31} + X_{32} + X_{33}$$

(cette transformation est analogue à la factorisation à droite).

L'application de (2) donne la règle

$$Y_1 \to cY_5 \mid hY_6$$

et les équations

$$Y_5 = X_{31} + X_{32}$$
$$Y_6 = X_{31} + X_{33}$$

L'équation (3) et les règles X_2 et X_4 de G1 donnent :

$$Y_3 \to e \mid dY_7 \mid fY_8 a$$

et les équations :

(7) $Y_7 = X_2 + X_4 = Y_3$

(8) $Y_8 = X_1 = Y_1$

d'où $\underline{Y_3 \to e \mid dY_3 \mid fY_1 a}$

Et ainsi de suite ... On obtient finalement :

G2 :

$$Y_o \to Y_1 Y_1 \mid aY_3$$
$$Y_1 \to Y_4 b \mid cY_5 \mid hY_6$$
$$Y_6 \to Y_3 Y_1 \mid g$$
$$Y_3 \to dY_3 \mid fY_1 a \mid e$$
$$Y_4 \to Y_3 Y_1 \mid bc \mid g$$
$$Y_5 \to Y_3 Y_1 \mid bc$$

L'intérêt de cette méthode est que le nombre de règles est réduit et que le nombre de symboles non-terminaux est réduit entre G1 et G2. On peut donc l'utiliser pour réduire une grammaire si l'on veut conserver la structure.

Certaines transformations rendent LL(1) des règles qui ne peuvent être transformées par SID.

L'exemple (3.2.7) :

$A \rightarrow B \mid C$

$B \rightarrow aB \mid b$

$C \rightarrow aC \mid c$

donne : $X \rightarrow Y$

et $Y = B + C$

$Y \rightarrow aZ \mid b \mid c$

et $Z = B + C = Y$

la grammaire structurellement équivalente obtenue est :

$X \rightarrow Y$

$Y \rightarrow aY \mid b \mid c.$

3.4. Méthodes particulières

3.4.1. Cas d'une règle qui ne satisfait pas à la condition LL(1) n° 3.

La grammaire de "word" et "number" (3.2.7) a été transformée en une grammaire LL(1) par un moyen identique à celui utilisé pour la syntaxe d'un bloc en Algol [Gri2].

bloc → begin declist ; stlist end

declist → dec dectail

dectail → ε | ; declist

stlist → st stail

stail → ε | ; stlist

On voit que Premier (; declist) et suivant (dectail) sont égaux à ";". Alors le ";" figurant dans la partie droite de "bloc" est supprimé puis mis à la place de la partie droite "ε" de "dectail" :

bloc → begin declist stlist end
declist → dec dectail
dectail → ; | ; declist

il faut factoriser : dectail → ; X
X → ε | declist

Cette méthode est valable à la condition que le non-terminal "declist" n'apparaisse pas ailleurs avec un autre contexte à droite que ";". Cette condition se traduirait pour la grammaire initiale par :

suivant (declist) = {;}

Si elle n'était pas réalisée, le langage généré ne serait plus le même et il faudrait apporter d'autres modifications pour assurer, si possible, la conservation du langage. Dans le cas plus général de règles :

$X \rightarrow \alpha Y Z \beta$
$Y \rightarrow \varepsilon \mid a\gamma$

où le terminal a appartient à suivant (Y), il faudrait substituer Z dans la partie droite de X afin de faire apparaître le symbole a comme suivant immédiatement Y. On appliquerait ensuite la transformation utilisée dans l'exemple qui précède à condition que :

<u>suivant (Y) = {a}</u>

Cette méthode nécessiterait l'usage d'une <u>substitution dépendante du contexte</u> (différente de la substitution à gauche) et pourrait être réalisée par un programme.

Mais l'intérêt de cette réalisation est diminué par de nombreux inconvénients.

La condition suivant (Y) = {a} est évidemment très restrictive.

La substitution dépendante du contexte ne conserve pas la propriété LL(1) d'après un théorème déjà cité (2.4.2). Autrement dit si $X \to \alpha Y Z \beta$ est une règle LL(1), la substitution de Z peut la changer en une règle non LL(1).

Enfin l'exemple qui suit montre qu'il n'est pas toujours possible d'obtenir une grammaire satisfaisant à la condition LL(1) n° 3.

La grammaire citée par Kurki-Suonio [Kur2] :

$$S' \to S \vdash \vdash$$
$$S \to aSA \mid \varepsilon$$
$$A \to abS \mid c$$

est LL(2) mais n'est pas LL(1) car la règle S ne satisfait pas à la condition n° 3. De plus le langage généré par cette grammaire n'est pas LL(1). [Kur2]. Il est donc impossible de trouver une grammaire LL(1) équivalente.

3.4.2. Cas de règles ne satisfaisant pas à la condition 2

Il s'agit de règles telles que :

$$A \to \alpha_i \mid \alpha_j$$

avec

$$(1) \quad \begin{array}{l} \alpha_i \overset{*}{\to} \omega B \psi_1 \\ \alpha_j \overset{*}{\to} \omega C \psi_2 \end{array}$$

et

$$(2) \quad \begin{array}{l} B \overset{+}{\to} \alpha B \beta \\ C \overset{+}{\to} \alpha C \gamma \end{array}$$

On a vu qu'elles n'étaient pas transformées par SID (§ 3.2.7).

Cependant dans tous les cas où β et γ sont égaux à la chaîne vide, on peut obtenir une règle LL(1) en se ramenant à la configuration :

$$\begin{array}{l} A \to \omega X \\ X \to \alpha Y \\ Y \to \alpha Y \mid \beta_1 \psi_1 \mid \gamma_1 \psi_2 \end{array} \qquad \text{avec} \begin{cases} B \to \beta_1 \\ C \to \gamma_1 \end{cases}$$

Mais si au moins β ou γ n'est pas égale à la chaîne vide alors la transformation en LL(1) n'est pas toujours possible comme le montre l'exemple de la fin du paragraphe 3.2.7.

Cette transformation est possible quand la méthode de Paull et Unger l'effectue. cf [Pau]. Cette méthode marche en particulier si β et γ sont égaux dans la condition (2) :

$$\begin{array}{l|l} A \to \omega B \mid \omega C & X \to \omega Y \\ B \to \alpha B \beta \mid \beta_1 & Y \to \alpha Y \beta \mid \beta_1 \mid \gamma_1 \\ C \to \alpha C \beta \mid \gamma_1 & \end{array}$$

Nous avons été ainsi amenés au problème de la caractérisation de familles de langages qui ne sont pas LL(k) pour $k \geq 1$. Plus précisément, nous avons étudié, en collaboration avec Henri Saya, la famille des langages de la forme $L_1 \cup L_2$ où :

$$L_1 = \{uv^i w_1 x_1^i z_1 = q : i > 0\}$$
$$L_2 = \{uv^j w_2 x_2^j z_2 = r : j > 0\}$$

où $u, w_1, w_2, x_1, z_1, z_2$ appartiennent à V_T^*
v, x_1 " " $V_T^* - \{\varepsilon\}$

L_1 et L_2 sont <u>deux ensembles infinis</u> dont <u>l'intersection est vide</u> et <u>pour toute chaîne q il existe une chaîne r telle que j = i et réciproquement</u>.

Cette étude, où nous donnons des conditions nécessaires et suffisantes pour que ces langages soient non LL(k) pour $k \geq 1$, doit faire l'objet d'un article. [Bor3].

On peut remarquer que cette famille a des représentants dans différentes classes de langages.

$\{a^n c b^n\} \cup \{a^n d b^n\}$ est <u>LL(1)</u>

$\{a^n b^n\} \cup \{a^n c^n\}$ et $\{a^n c b^n c\} \cup \{a^n d b^n d\}$ sont des langages qui sont <u>non LL(k) quel que soit k</u>, mais qui sont <u>déterministes</u> [Ros2].
$\{a^n b^n\} \cup \{a^n b^{2n}\}$ est <u>non déterministe</u> [Gin4] et non ambigu.

CHAPITRE IV

ETUDE DE LA NOTATION ULD

CHAPITRE IV

ETUDE DE LA NOTATION ULD

4. ETUDE DE LA NOTATION ULD

4.1.

On a donné dans le paragraphe 1.6 les éléments de base de la notation ULD. On verra qu'elle permet d'aborder d'une manière différente le problème de la mise sous forme LL(1) d'une grammaire CF. Cependant ULD présente d'autres avantages relativement à l'analyse syntaxique. Ce chapitre 4 répond à deux buts principaux.

Le premier est la transformation d'une grammaire ULD en une grammaire BNF. Les fonctions sémantiques pouvant être introduites dans une grammaire ULD, on peut ainsi utiliser l'ULD dans le système d'écriture de compilateurs de M. Griffiths et M. Peltier.

Le second est la définition des conditions nécessaires et suffisantes pour qu'une grammaire ULD soit LL(1). Ces conditions permettent d'envisager la génération d'un analyseur optimisé obtenu directement à partir d'ULD.

Cette étude de la notation ULD fait suite au travail de M. Assabgui [Ass].

4.2. Définitions

On définit ici d'une manière plus formelle les possibilités offertes par la notation ULD.

On a les équivalences suivantes :

4.2.1.

$$V \rightarrow \alpha_1\beta_1\alpha_2 \mid \ldots \mid \alpha_1\beta_n\alpha_2$$

est équivalent à :

$$V \rightarrow \alpha_1 < \beta_1 \mid \ldots \mid \beta_n > \alpha_2$$

4.2.2.

$$V \rightarrow \alpha_1\alpha_2 \mid \alpha_1\beta_1\alpha_2 \mid \ldots \mid \alpha_1\beta_n\alpha_2$$

est équivalent à :

$$V \rightarrow \alpha_1 (\beta_1 \mid \ldots \mid \beta_n) \alpha_2$$

4.2.3.

$$V \rightarrow U \mid VU \quad \text{ou} \quad V \rightarrow U \mid UV$$

est équivalent à :

$$V \rightarrow U +$$

4.2.4.

$$V \rightarrow \alpha_1\beta_1 (< \beta_2\beta_1 > +) \alpha_2$$

est équivalent à :

$$V \rightarrow \alpha_1 <.\beta_2.\beta_1.> \alpha_2$$

4.2.5.

$$V \rightarrow \alpha_1 (< \beta_2.\beta_1.>) \alpha_2$$

est équivalent à :

$$V \rightarrow \alpha_1 (.\beta_2.\beta_1.) \alpha_2$$

Dans la notation des listes (4.2.4 et 4.2.5) β_2 doit être un symbole terminal, β_1 est soit un terminal, soit un non-terminal. Ailleurs U, V, α, β ont le sens habituel.

Mais ULD autorise une expression écrite à l'aide du métalangage à l'intérieur d'une autre. La définition d'ULD est donc récursive.

Une règle ULD sera écrite selon la métagrammaire :

```
RULE → classname → DEFINITION
DEFINITION → SEQUENCE | SEQUENCE | DEFINITION
SEQUENCE → UNIT | UNIT SEQUENCE
UNIT → basicname | basicname + |
       classname | classname + |
       < DEFINITION > | < DEFINITION > + |
       ( DEFINITION ) | ( DEFINITION ) + |
    <.basicname.UNIT.> | (.basicname.UNIT.)
```

basicname représente un symbole terminal quelconque
classname représente un symbole non-terminal quelconque

On a souligné les métasymboles ULD dans le cas où il pouvait y avoir confusion avec les méta-métasymboles → et |.

Cette métagrammaire est restrictive par rapport à celle définie dans [Urs], mais la puissance de la notation n'est pas diminuée.

4.3. Cas des règles récursives

L'équivalence 4.2.3 peut être généralisée.
Soit par exemple :

$$V_1 \rightarrow a \mid V_1 s \mid V_2 t$$
$$V_2 \rightarrow b \mid V_1 u \mid V_2 v$$

D'après le paragraphe 2.2.2 il y a équivalence entre $X \rightarrow XY \mid Z$ et $X \rightarrow Z(Y)+$ et par une méthode de résolution de système d'équations algébrique [Sal] on obtient :

$V_1 \rightarrow$ < a | < b | au > (tu | v) + t >

(s | su (tu | v) + t) +

$V_2 \rightarrow$ < b | < a | < b | au > (tu | v) + t >

(s | su (tu | v) + t) + u > (v) +

On a par conséquent un moyen d'éliminer la récursivité à gauche, différent de celui étudié au paragraphe 2.2, en utilisant cette méthode algébrique.

On voit qu'ULD permet d'effectuer le calcul d'expressions régulières sur le vocabulaire d'une grammaire.

Il existe des algorithmes pour la construction d'une expression régulière à partir d'un graphe [McN, Brz].

Une autre conséquence est qu'on ne peut pas représenter avec une expression régulière sur V un langage comme $\{a^n b^n : n > 0\}$ engendré par une règle auto-emboitée

$S \rightarrow aSb \mid ab$

et qui ne peut pas être engendré par une grammaire d'états finis.

4.4. Avantages de la notation ULD

La notation ULD offre la possibilité de factoriser les éléments de parties droites (4.2.1 & 4.2.2), d'écrire les règles récursives à droite ou à gauche d'une manière condensée (4.2.3) et enfin d'écrire simplement les règles générant des listes (4.2.4 & 4.2.5). Elle possède même une souplesse plus grande que les expressions régulières. L'opération étoile peut être écrite soit (a) +, soit (a +).

La grammaire d'un langage peut donc être écrite plus facilement :

- diminution du nombre de règles
- raccourcissement des règles.

En utilisant judicieusement ULD on peut aussi faciliter la compréhension d'une grammaire en mettant en évidence les éléments du langage.

La grammaire permet de fabriquer un analyseur : la diminution de la grammaire se traduit par un gain d'espace pour l'analyseur.

Enfin les factorisations et l'écriture des récursivités à gauche permettent d'écrire plus aisément une grammaire satisfaisant aux conditions LL(1). C'est ce que nous allons voir dans les paragraphes suivants.

4.5. Traduction de ULD en BNF

Il est clair qu'une grammaire ULD peut être réécrite en une grammaire BNF équivalente en utilisant par exemple les conventions 4.2.1, 2, 3, 4, 5. L'ensemble des langages générés par une grammaire ULD est donc identique à celui obtenu par BNF ou context-free. Mais, afin de définir des conditions LL(1) pour une grammaire ULD, nous allons adopter la démarche suivante : on dira qu'une grammaire ULD est LL(1) si et seulement si la grammaire BNF équivalente est LL(1). Il faut pour cela que les équivalences utilisées ne soient pas triviales, c'est-à-dire qu'on ne doit pas obtenir à partir d'ULD une forme de règle BNF qui serait nécessairement non LL(1).

Par exemple si nous traduisons :

$$V \rightarrow a\ +$$

en

$$V \rightarrow a \mid aV$$

la règle obtenue ne satisfait pas à la condition LL(1) n° 2 mais :

$$V \rightarrow aX$$
$$X \rightarrow \varepsilon \mid aX$$

sont des règles équivalentes et LL(1).

Nous avons trouvé des règles BNF simples satisfaisant à ce critère et réalisant l'équivalence. Voir le tableau ci-dessous.

Remarquons que les équivalences définies ne sont pas optimales en ce sens que, dans certains cas particuliers, la traduction de règles ULD en BNF peut conduire à des règles redondantes.

Par exemple : $V \rightarrow x\ (\ a\ +\ )\ y$

se traduira en

$$V \rightarrow xXy$$
$$X \rightarrow \varepsilon \mid aY$$
$$Y \rightarrow \varepsilon \mid aY$$

alors que la traduction la plus simple est

$$V \rightarrow xXy$$
$$X \rightarrow \varepsilon \mid aX$$

Mais on a introduit cette simplification dans le traducteur.

Dans le tableau qui suit, X et Y représentent des nouveaux symboles non-terminaux créés par le traducteur. En 4.5.3, U peut représenter soit un terminal, soit un non-terminal.

	ULD	BNF
4.5.1.	$V \to \alpha_1 < \beta_1 \mid \ldots \mid \beta_n > \alpha_2$	$V \to \alpha_1 X \alpha_2$ $X \to \beta_1 \mid \ldots \mid \beta_n$
4.5.2.	$V \to \alpha_1 (\beta_1 \mid \ldots \mid \beta_n) \alpha_2$	$V \to \alpha_1 X \alpha_2$ $X \to \varepsilon \mid \beta_1 \mid \ldots \mid \beta_n$
4.5.3.	$V \to \alpha_1 U + \alpha_2$	$V \to \alpha_1 U X \alpha_2$ $X \to \varepsilon \mid UX$
4.5.4.	$V \to \alpha_1 < \beta_1 \mid \ldots \mid \beta_n > + \alpha_2$	$V \to \alpha_1 X Y \alpha_2$ $X \to \beta_1 \mid \ldots \mid \beta_n$ $Y \to \varepsilon \mid XY$
4.5.5.	$V \to \alpha_1 (\beta_1 \mid \ldots \mid \beta_n) + \alpha_2$	$V \to \alpha_1 Y \alpha_2$ $Y \to \varepsilon \mid XY$ $X \to \beta_1 \mid \ldots \mid \beta_n$
4.5.6.	$V \to \alpha_1 <.a.\beta.> \alpha_2$	$V \to \alpha_1 \beta X \alpha_2$ $X \to \varepsilon \mid a\beta X$
4.5.7.	$V \to \alpha_1 (.a.\beta.) \alpha_2$	$V \to \alpha_1 X \alpha_2$ $X \to \varepsilon \mid \beta Y$ $Y \to \varepsilon \mid a\beta Y$

4.6. Conditions LL(1) pour une grammaire ULD

Une grammaire ULD peut contenir des règles BNF ordinaires. Pour ces règles les conditions LL(1) sont celles définies en 1.5.1. Il reste à étudier les formes de base traduites dans le paragraphe précédent. On énoncera d'abord les conditions LL(1) sur les règles BNF. On montrera ensuite que ces conditions ne dépendent pas des nouveaux symboles non-terminaux introduits.

La condition 1 est la même dans tous les cas : il ne doit pas y avoir de symboles non-terminaux récursifs à gauche.

Cas 4.5.1 :

condition 2: Premier $(\beta_i) \cap$ Premier $(\beta_j) = \emptyset \quad i \neq j$
condition 3: si $\beta_i \overset{*}{\rightarrow} \varepsilon$ alors Premier $(\beta_j) \cap$ suivant $(X) = \emptyset$
condition 4: au plus un β_i peut dériver sur la chaîne vide

Cas 4.5.2 :

condition 2: Premier $(\beta_i) \cap$ Premier $(\beta_j) = \emptyset \quad i \neq j$
condition 3: Premier $(\beta_i) \cap$ suivant $(X) = \emptyset$
condition 4: aucun β_i ne dérive sur la chaîne vide

Cas 4.5.3 :

condition 3: Premier $(U) \cap$ suivant $(X) = \emptyset$
condition 4: U ne dérive pas sur la chaîne vide

Cas 4.5.4 :

condition 2: Premier $(\beta_i) \cap$ Premier $(\beta_j) = \emptyset \quad i \neq j$
condition 3: premier $(X) \cap$ suivant $(Y) = \emptyset$
condition 4: aucun β_i ne dérive pas sur la chaîne vide

Cas 4.5.5 :

les conditions 2, 3, 4 sont les mêmes que dans le cas 4.5.4.

Cas 4.5.6 :

condition 3: a n'appartient pas à suivant (X)

Cas 4.5.7 :

condition 3: a n'appartient pas à suivant (Y) = suivant (X).
Premier (β) n'appartient pas à suivant (X)
condition 4: β ne dérive pas sur la chaîne vide.

Toutes ces conditions sont indépendantes des symboles non- terminaux introduits.

Dans tous les cas, si V n'est pas récursif à gauche, alors aucun symbole X ou Y n'est récursif à gauche. En effet à cause des autres conditions LL(1) (en particulier la condition 4), les seuls cas où X (ou Y) seraient récursifs à gauche impliqueraient que $V \ell X$ et $X \ell^{+} V$, c'est-à-dire V récursif à gauche.

Dans les cas 4.5.1, 2, 3, 6, 7
suivant (X) = si $\alpha_2 \overset{*}{\rightarrow} \varepsilon$ alors Premier $(\alpha_2) \cup$ suivant (V)
sinon Premier (α_2)

Dans les cas 4.5.4, 5 suivant (Y) a la même expression que suivant (X) et premier (X) = $\bigcup_i$ Premier (β_i).

4.7. Programme de traduction

Le programme de traduction de ULD en BNF lit la grammaire ULD d'entrée, signale les erreurs dans l'écriture des règles, liste la grammaire ULD et génère les règles BNF. Ces opérations sont effectuées en un seul passage. Ensuite la grammaire BNF est listée avec la table de références croisées des symboles (fonctions, terminaux, non-terminaux).

Le principe du traducteur est un analyseur dirigé par la metagrammaire, comportant des instructions intercalées qui construisent les règles BNF sous forme de listes. Pour calculer ces règles on utilise deux piles. Le sommet de chaque pile contient respectivement le pointeur de la partie droite (1) courante (c'est-à-dire la partie droite en cours d'évaluation)

---

(1) On appelle 'partie droite' un terme α_i dans la règle
$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_i \mid \ldots \mid \alpha_n$

PROGRAMME ULD : MESSAGES D'ERREUR

```
INPUT SYNTAX=
 1    PROGRAM          PROCEDURECL F1 +
/// METASYMBOL + AFTER FUNCTION NAME
                      +
/// ILLEGAL METASYMBOL + OR .

 2    PROCEDURECL      ( PREFIXLIST LABELLIST 'PROCEDURE' ( PARAMETERLIS ( PROCEDUREOPT 'SEMICOLON'
                       SENTENCELIST
/// EXPECTED METASYMBOL ) IS NOT FOUND

/// EXPECTED METASYMBOL ) IS NOT FOUND

/// EXPECTED METASYMBOL ) IS NOT FOUND

 3    PARAMETERLIS     'LBRACK' < . F2
/// SYMBOL USED AS SEPARATOR IS NOT A BASIC SYMBOL
                      'COMMA'
/// METASYMBOL . OMITTED AFTER SEPARATOR
                      IDENTIFIER
/// METASYMBOL . OMITTED AFTER UNIT

/// METASYMBOL > OMITTED AFTER METASYMBOL .
                      'RBRACK'
 4    SENTENCELIST     < SENTENCE + > >
/// ILLEGAL METASYMBOL > OR )

                      ENDCLAUSE
 5    ENDCLAUSE        < . PREFIXLIST > < LABELLIST . > 'END' 'SEMICOLON'
 6    SENTENCE         PROCEDURECL . ENTRYCL . DECLARATIONS . FORMATSENTEN . STATEMENT ;
```

```
 7    ENTRYCL          LABELLIST .
/// ILLEGAL METASYMBOL + OR .
                    'ENTRY' ( PARAMETERLIS , ) ( RETURNSATTRI ) 'SEMICOLON'
 8     /// END OF ULD GRAMMAR
 18 PREFIXLIST IS UNDEFINED CLASSNAME
 19 LABELLIST IS UNDEFINED CLASSNAME
 21 PROCEDUREOPT IS UNDEFINED CLASSNAME
 23 IDENTIFIER IS UNDEFINED CLASSNAME
 27 DECLARATIONS IS UNDEFINED CLASSNAME
 28 FORMATSENTEN IS UNDEFINED CLASSNAME
 29 STATEMENT IS UNDEFINED CLASSNAME
 30 RETURNSATTRI IS UNDEFINED CLASSNAME
ULD GRAMMAR:LAST CLASSNAME= 30
BNF GRAMMAR : LAST CLASSNAME= 42

LAST CLASSNAME IS FALSE IN THE FOURTH DATUM
START NEW JOB IF ANY
IHE140I FILE SYSIN - END OF FILE ENCOUNTERED AT OFFSET +0015A FROM ENTRY POINT MAINPRO
```

et le pointeur de la règle courante. Chaque fois qu'une partie droite est calculée, elle est ajoutée à la règle courante. Chaque fois qu'une règle est calculée elle est sortie sur disque (le pointeur 'règle' et le pointeur 'partie droite' sont alors désempilés). Les conventions sur la grammaire d'entrée sont compatibles avec celles de |Gri3| :

→ est représenté implicitement par des espaces

| est représenté par la virgule,

la fin d'une règle est indiquée par une étoile *

Les autres métasymboles sont ceux définis dans le paragraphe 1.6.

De plus, comme ULD ne prévoit pas la chaîne vide dans l'écriture des règles, cette dernière possibilité a été introduite. Par exemple il est équivalent d'écrire :

(LABELLIST) ou « | LABELLIST » ou « LABELLIST | »

Nous offrons une possibilité nouvelle, celle d'écrire des commentaires dans le texte de la grammaire d'entrée. Ces commentaires sont listés, mais sont ignorés par le traducteur. (voir § 5.3).

On peut placer des noms de fonctions sémantiques dans la grammaire ULD aux conditions suivantes :

- un nom de fonction ne peut pas être suivi du métasymbole"+"
- un nom de fonction ne peut pas être contigu au métasymbole"."

L'utilisation des métasymboles « (et) » permet de respecter ces conditions.

Exemple

PROGRAM → f_1 PROCEDURE f_2 +

n'est pas correct,
on écrira :

PROGRAM → f_1 « PROCEDURE f_2 » +

Une fois la grammaire BNF obtenue, on peut éventuellement effectuer des transformations sur cette grammaire, puis tester les conditions LL(1) et générer l'analyseur dans le langage de macroinstructions [Gri5]. On peut ainsi obtenir un analyseur à partir d'une syntaxe ULD.

4.8. Génération d'un analyseur directement à partir d'ULD

La notation ULD présente de nombreux avantages (4.4). Elle permet aussi (4.7) l'introduction des fonctions sémantiques. Toutefois il est difficile d'évaluer le gain d'efficacité pour l'analyseur (par rapport à l'analyseur obtenu à partir d'une grammaire qu'on aurait écrite directement en BNF).

Un gain important serait obtenu en générant l'analyseur directement à partir de la grammaire ULD, en utilisant les conditions du paragraphe 4.6 et les idées de M. Assabgui [Ass] sur la génération nouvelle de l'analyseur. Le principe est le suivant : On a déjà dit que la metagrammaire ULD permet d'écrire des expressions régulières sur le vocabulaire d'une grammaire (on peut faire le rapprochement avec d'autres formalismes [Bou, Tix]. Ces expressions correspondent, du point de vue de l'analyse, à un automate d'états finis. Donc ces expressions doivent générer des actions du type branchement conditionnel ou inconditionnel (macros DECIDE, GOTO), tandis que les actions du type appel récursif de routine (macro CALL) correspondent à un automate à pile.

On remplacerait donc des actions du deuxième type par des actions du premier type dans la nouvelle génération, d'où un gain de temps et d'espace pour l'analyseur.

Remarque

Actuellement la rapidité de l'analyseur produit est de l'ordre de celle des moyens d'accès les plus rapides; elle n'a pas besoin d'être beaucoup améliorée. C'est le gain de place qui est intéressant.

Exemple

La règle $V \rightarrow \alpha_1 U + \alpha_2$

donne dans le système actuel :

$$V \rightarrow \alpha_1 UX\alpha_2$$
$$X \rightarrow \varepsilon \mid UX$$

et le programme d'analyse :

```
ROUTINE V
   :
   CALL U
   CALL X
   :
RETURN
ROUTINE X
   DECIDE L1,premier (U)
   EXIT
L1 CALL U
   ENTER X
RETURN
```

tandis qu'on obtiendrait à partir d'ULD :

```
ROUTINE V
   :
L1 CALL U
   DECIDE L1,premier (U)
   :
RETURN
```

Si U est un symbole terminal a, alors on génère CHECK a à la place de CALL U.

CHAPITRE V

PROGRAMMES

CHAPITRE V

PROGRAMMES

5. PROGRAMMES

5.1. Les programmes de transformation de grammaire

Les programmes suivants ont été écrits :

- élimination de la récursivité à gauche
suivant l'algorithme présenté à la fin du paragraphe 2.2.3
- substitutions-factorisations
suivant l'algorithme présenté en § 3.2.3 et 3.2.4
- traduction d'une grammaire ULD en une grammaire BNF
suivant les paragraphes 4.5 et 4.7.

Ces programmes ont été écrits dans le langage PL/1 et sont disponibles sous forme de texte source ou de module chargeable sur IBM 360.

En fait nous avons en tout cinq programmes. Chaque programme correspond à un ensemble de transformations (voir le schéma sur la Figure 4). Le découpage a été choisi de manière à offir de multiples possibilités (lesquelles sont figurées par des flèches sur la Figure 4). Le découpage évite aussi d'avoir de trop gros programmes. Pour nos programmes 150 K octets sont suffisants pour traiter une grammaire de grosseur moyenne, telle que celle d'ALGOL 60. La transmission des données entre deux programmes se fait automatiquement, à l'aide des disques.

Le transformateur de grammaire original a été coupé en deux programmes et correspond à la configuration BNF-TGA de la Figure 4.

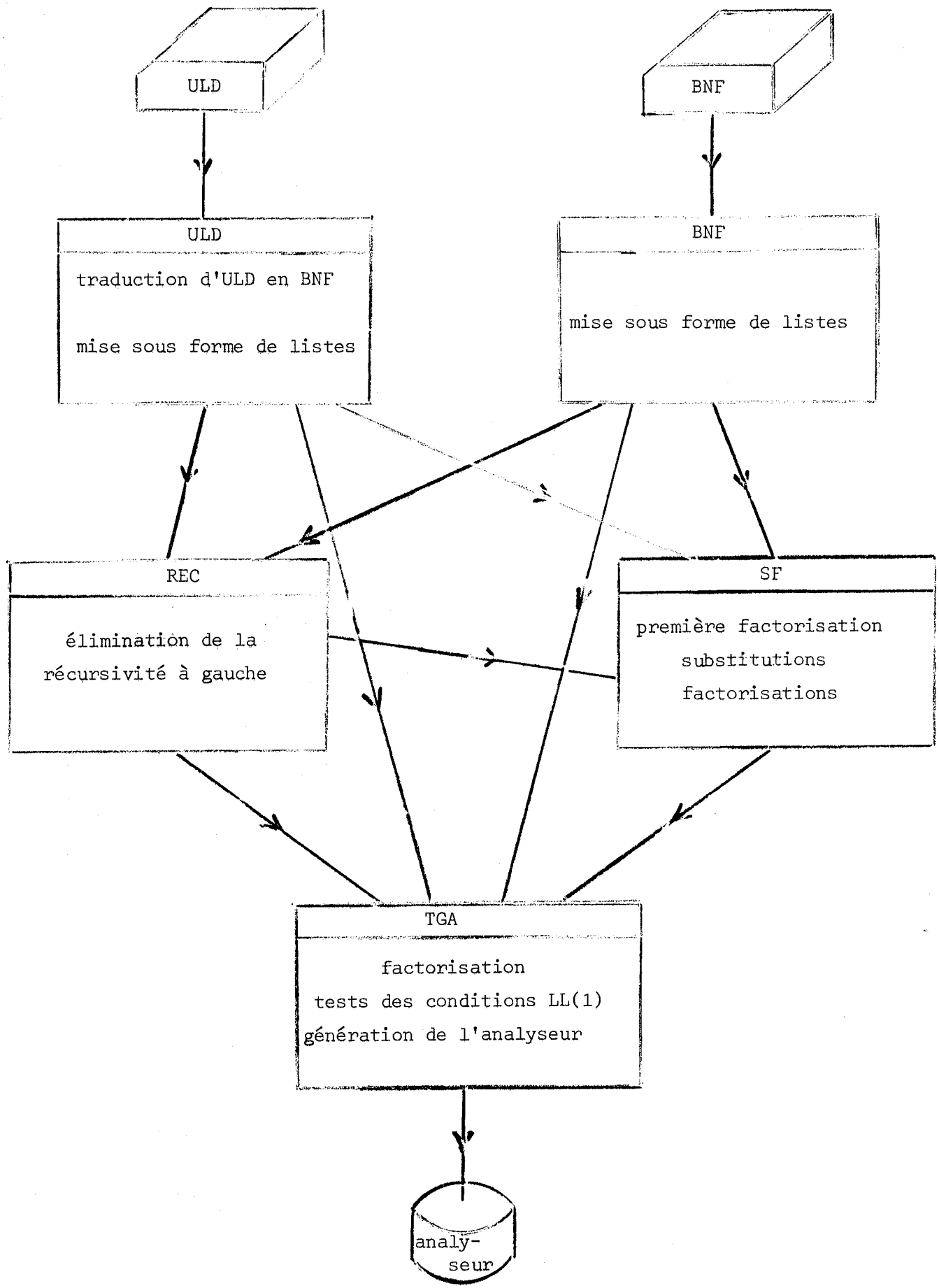

ULD
BNF
ULD
traduction d'ULD en BNF
mise sous forme de listes
BNF
mise sous forme de listes
REC
élimination de la
récursivité à gauche
SF
première factorisation
substitutions
factorisations
TGA
factorisation
tests des conditions LL(1)
génération de l'analyseur
analy-
seur

Figure. 4

Toutes les conventions choisies assurent la compatibilité de ces programmes. En particulier nous avons jugé plus simple de continuer à écrire les messages imprimés en langue anglaise.

5.2. Outils des programmes

5.2.1. Gestion de listes

Chaque programme de la Figure 4 utilise un ensemble de procédures de gestion de listes (d'abord écrites en ALGOL, puis traduites en PL/1 par un programme de M. Berthaud) dont la description se trouve dans [Gri1].

Nous avons simplement ajouté certaines procédures, comme la classique fonction 'append' :

```
append(x, y) = si x = nil alors cons(y, nil)
               sinon cons(hd(x), append(tl(x), y)).
```

Nous avons aussi ajouté la déclaration de tableaux de listes à l'aide, par exemple, de la procédure (PL/1) :

```
DCLISTARRAY : PROCEDURE (P) ; DCL P ENTRY ;
DCL I FIXED BINARY (15) ;
DO I = 1 BY 1 TO XXX ;
CALL P (LISTARRAY (I)) ;
END ;
END ;
```

et de la déclaration :

```
DCL LISTARRAY (1 : XXX) FIXED BINARY (31) ;
```

Remarquons que les identificateurs se réfèrant à des listes ne peuvent être que statiques pour la partie du programme qui les utilise. Les tableaux de listes doivent donc être à bornes fixes. Dans l'exemple ci-dessus l'identificateur XXX doit se référer à une valeur fixée à l'avance. On doit prévoir des bornes suffisamment grandes. Heureusement en général on n'a besoin que de tableaux de listes ayant de petites dimensions.

Dans l'élimination de la récursivité à gauche la dimension dépend de l'ordre de la récursivité (c'est-à-dire le nombre d'éléments de la clique maximale). Pratiquement cet ordre est rarement supérieur à 1. Dans la traduction d'ULD en BNF la dimension des tableaux de listes dépend du degré d'emboîtement des symboles ( ) et < > qui n'est jamais très grand.

On demande de fournir dans les données du programme REC la borne supérieure pour ces tableaux.

On indique dans un tableau les données sur cartes perforées nécessaire à chaque programme.

programme	données
BNF	top printc grammaire
ULD	top printc lastclassname grammaire
REC	top printc dimension
SF	top printc number
TGA	top printc offset

top-2052 — est égal à la dimension de la zone de liste et on prend en général top = 22000.

printc — si printc n'est pas égal à zéro, l'activation du ramasse-miettes (collect garbage) est signalée. On prend en général printc = 0.

dimension — dimension demandée pour les tableaux de listes.
pour REC dimension = 5 est suffisant dans la majorité des cas.

grammaire — le paquet de cartes doit comporter successivement :
le chiffre 1 suivi de la liste des fonctions sémantiques
le chiffre 2 suivi de la liste des symboles terminaux
le chiffre 3 suivi des règles (voir § 5.3)
le chiffre 4.

lastclassname — numéro du dernier symbole non-terminal, égal à :
nombre de fonctions + nombre de symboles terminaux + nombre de règles.
(si cette donnée est fausse, un message l'indiquera).

number — nombre d'itérations demandé par l'utilisateur pour l'algorithme § 3.2.4.

offset — numéro de la première routine de l'analyseur généré.
en général offset = 700.

Remarque

La version 5 du compilateur PL/1 pour le 360 réserve un demi-mot, pour une variable déclarée avec l'attribut FIXED (15). Précedemment c'était un mot complet qui était réservé. On pourrait modifier la gestion des listes de manière à avoir une zone de listes en mémoire centrale occupant moins de place. En effet 15 bits sont suffisants pour les éléments des tableaux HD et TL.

5.2.2. Représentation de la grammaire

Une fois que la grammaire d'entrée a été lue sur cartes, les règles de la grammaire sont toujours traitées sous forme de listes et les symboles sont représentés par des nombres entiers. La grammaire est transmise à l'aide des disques. Les informations suivantes sont aussi transmises automatiquement d'un programme à un autre :

- le numéro (code) du premier symbole terminal
- le numéro (code) du premier symbole non-terminal
- le numéro (code) du dernier symbole non-terminal

5.2.3. Calcul des relations fondamentales ℓ^+ et s

Les relations fondamentales sont calculées, on l'a déjà dit, à l'aide des matrices booléennes. Nous avons pensé utiliser les chaînes de bits et les fonctions incorporées pour le traitement de ces chaînes disponibles dans le langage PL/1. On obtient des programmes simples à écrire, mais beaucoup moins performants que si l'on effectue le calcul sur des entiers. Les matrices booléennes sont donc représentées par des tableaux dont les éléments sont des mots (attribut FIXED BINARY (31)).

On utilise les algorithmes de calcul de fermeture transitive présentés dans [Gri2], plus rapides que l'algorithme de Warshall [War] pour des matrices assez creuses, comme c'est généralement le cas pour des grammaires de langages de programmation.

5.3. Forme de la grammaire d'entrée

5.3.1. Programme BNF

La grammaire d'entrée est perforée sur cartes selon les conventions [Gri3] :

BNF	EBCDIC
→	espace (s)
|	,
fin de règle	*
symbole (terminal ou non-terminal)	chaîne de caractères alphanumériques (les caractères sont ignorés après le 12ème)
opérateur de concaténation (implicite)	espace (s)

Le format est librement choisi.

5.3.2. Programme ULD

Pour une grammaire ULD les conventions sont les mêmes qu'en § 5.3.1 et on utilise en plus les caractères EBCDIC suivants :

<	>	(	)	.	+

(voir § 1.6).

Le programme ULD (voir § 4.7) liste la grammaire ULD. Afin de clarifier le listage de cette grammaire, nous avons introduit la possibilité de sauts de lignes sur l'imprimante, à l'aide du caractère % que l'on peut placer à tout endroit de la grammaire. On placera ce caractère par exemple après les virgules si on veut séparer les différentes alternatives.

Nous avons rendu possible l'inclusion de commentaires dans la grammaire, commentaires apparaissant dans le listage de la grammaire ULD.

Ces commentaires sont imprimés dans un cadre, à 80 caractères par ligne.

début de commentaire	@
fin de commentaire	&
saut de ligne	%
commentaire	peut être composé de n'importe quel caractère sauf &

Un commentaire est imprimé comme image de cartes perforées, ce qui permet une certaine liberté dans l'édition du commentaire.

L'inclusion de commentaires est intéressante pour plusieurs raisons :

- décomposition de la grammaire en chapitres à l'aide de titres
- inclusion de texte concernant la syntaxe ou la sémantique du langage
- les symboles étant coupés à 12 caractères, on peut indiquer avec précision la signification de tel ou tel symbole.

```
        INPUTGRAMMAR: BEGIN;
DECLARE OUT ENTRY(FIXED BINARY(31)),                                    1
OUT:PROCEDURE(A);
DECLARE A FIXED BINARY(31);
/*'VALUE' A*/;
IF A=-7 THEN
CALL OUTPAR((LEFT));
ELSE IF A=-8 THEN
CALL OUTPAR((RIGHT));
ELSE CALL OUTITE((A));
END;
DECLARE REFSIZ FIXED BINARY(31);
DECLARE(INDEX$,STARTB,STARTF,STARTR,ENDRUL,ENDALT,NEWRUL,A,I,J,NAMELI,
IP,SAVE,LBRACK,RBRACK) FIXED BIN(31) ;
DECLARE(NAMETA(1:1256,1:3))FIXED BINARY(31);
DECLARE(BUFF(1:3))FIXED BINARY(31);
DCL(LEFTBRACE,RIGHTBRACE,ONEDOT,THREEDOTS,CNAME,NEWCLASS,LASTCLASS,
   ITEM,CRULE,CALT,TABCLASS(100)) FIXED BIN(31);
DCL (NAMECOUNT,CODE(1:400)) FIXED BIN(31);
DCL OUTPRINT BIT(1);
DCL (LCOMMENT,RCOMMENT,SCOMMENT) FIXED BIN(31);
DCL BACKLBRACK FIXED BIN(31);
DCL CHARCOUN FIXED BIN(31);
DECLARE PRINTN ENTRY(FIXED BINARY(31));
PRINTN:PROCEDURE(INDEX$);
DECLARE INDEX$ FIXED BINARY(31);
/*'VALUE' INDEX$*/;
BEGIN;
DECLARE(I,J,CHAR,WORD,UNIT)FIXED BINARY(31);
UNIT=OUTUNI;
OUTUNI=PRINT;
CALL OUTPUT(63);
      IF INDEX$< FIRSTC THEN IF INDEX$>= FIRSTB THEN
CALL OUTPUT(54);
DO I=1,2,3;
DO ;
WORD=NAMETA(INDEX$,I);
DO J=1,2,3,4;
DO ;
CHAR=EUNPAC(WORD);
IF CHAR=63 THEN
GOTO ENDNAM;
CALL OUTPUT((CHAR));
END;
END;
END;
END;
      ENDNAM: IF INDEX$<FIRSTC THEN IF INDEX$>= FIRSTB THEN
CALL OUTPUT(54);
OUTUNI=UNIT;
;
END/*PRINTNAME */;
END;
DECLARE FINDNA ENTRY RETURNS(FIXED BINARY(31));
```

```
FINDNA:PROCEDURE FIXED BINARY(31);
DECLARE ($RTURN,UNIT) FIXED BIN(31);
DO ;
IF CNAME ¬= 0 THEN
DO;
A=CNAME;
CNAME=0;
OUTPRINT='0'B;
GOTO EXIT;
END;
IF SAVE¬=0 THEN
DO ;
A=SAVE;
SAVE=0;
END;ELSE
STARTA : A = INPUT;
IF A = 63 THEN GOTO STARTA;
IF A = 51 | A = SCOMMENT THEN
DO ;
      IF NAMECOUNT = -1 THEN
      PUT EDIT(' ')(SKIP,COLUMN(5),A);
      ELSE DO;
      PUT EDIT(' ')(SKIP,COLUMN(21),A);
      NAMECOUNT=0; END;
      GOTO STARTA;
END ;
IF A = 47 | A = LCOMMENT THEN
DO ;
UNIT=OUTUNI;OUTUNI=PRINT;
CHARCOUNT=1;
PUT EDIT(' ')(SKIP(2),COLUMN(21),A);
DO I=1 BY 1 TO 90;
CALL OUTPUT(60);
END;
PUT EDIT('*','*','*     ')(SKIP,COLUMN(22),A,COLUMN(111),A);
CALL OUTPUT(63);
READAGAIN : A = INPUT;
      IF A ¬= 57 THEN
        DO;
        IF A = 51 THEN
          PUT EDIT('     *','*     ')(COLUMN(107),A,SKIP,COLUMN(22),A)
        IF CHARCOUNT < 80 THEN DO;
          IF A = 51 THEN CALL OUTPUT(63); ELSE
          CALL OUTPUT(A);
          CHARCOUNT=CHARCOUNT+1; END;
        ELSE DO;
          PUT EDIT('     *','*     ')(COLUMN(107),A,SKIP,COLUMN(22),A)
```

```
            IF A = 51 THEN CALL OUTPUT(63); ELSE
            CALL OUTPUT(A);
            CHARCOUNT=1; END;
          GOTO READAGAIN;
          END;
        ELSE
          DO;
            PUT EDIT('     *','*','     *','******')(COLUMN(107),A,SKIP,
                  COLUMN(22),A);
          DO I = 1 BY 1 TO 85;
          CALL OUTPUT(60);
          END;
          CHARCOUNT=0;
          OUTUNI=UNIT;
          IF NAMECOUNT = -1 THEN
          PUT EDIT(' ')(SKIP(2),COLUMN(5),A);
          ELSE DO;
          PUT EDIT(' ')(SKIP(2),COLUMN(21),A);
          NAMECOUNT=0; END;
          END;
GOTO STARTA;
END ;
      IF A=1 THEN A=STARTF;ELSE IF A=2 THEN A=STARTB;ELSE IF A=3 THEN
      A=STARTK;ELSE IF A=4 THEN A=ENDRUL;ELSE IF A=59 THEN A= ENDALT;
ELSE IF A = 60 THEN A = NEWRUL;
  ELSE IF A = 49 THEN A = LEFTBRACE ;
     ELSE IF A=50 THEN A = RIGHTBRACE ;
        ELSE IF A=61 THEN A=LBRACK ;
        ELSE IF A=62 THEN A=RBRACK ;
        ELSE IF A=58 THEN A=UNEDOT ;
        ELSE IF A = 56 THEN A = THREEDOTS;
          ELSE IF A = 47 THEN A = LCOMMENT;
          ELSE IF A = 57 THEN A =RCOMMENT;
          ELSE IF A = 51 THEN A = SCOMMENT;
IF A<0 THEN
GOTO EXIT;
DO I=1,2,3;
BUFF(I)=0;
END;
DO I=1,2,3;
DO J=1,2,3,4;
DO ;
BUFF(I)=256*BUFF(I)+A;
A=INPUT;
IF A = 60 THEN A = NEWRUL;
ELSE IF A=59 THEN A=ENDALT ;
  ELSE IF A=49 THEN A=LEFTBRACE;
```

(SUBRG) :

```
      ELSE IF A=50 THEN A=RIGHTBRACE;
         ELSE IF A=61 THEN A=LBRACK ;
         ELSE IF A=62 THEN A=RBRACK ;
         ELSE IF A=58 THEN A=ONEDOT ;
         ELSE IF A = 56 THEN A = THREEDOTS;
           ELSE IF A = 47 THEN A = LCOMMENT;
           ELSE IF A = 57 THEN A =RCOMMENT;
           ELSE IF A = 51 THEN A = SCOMMENT;
IF A=63 | A < 0 THEN GOTO L1 ;
END;
END;
END;
REPEAT : IF A ¬= 63 THEN IF A ¬= 60 THEN
  IF A ¬= 59 THEN IF A ¬= 49 THEN
     IF A ¬= 50 THEN IF A ¬= 61 THEN
        IF A ¬= 62 THEN IF A ¬= 58 THEN IF A ¬= 56 THEN
          IF A ¬= 47 THEN IF A ¬= 57 THEN IF A ¬= 51 THEN
DO ;
A=INPUT;
GOTO REPEAT;
END;
SAVE=A;
LOOK:DO I=1 BY 1 TO NAMELI;
DO ;
DO J=1,2,3;
IF BUFF(J)¬=NAMETA(I,J) THEN
GOTO NEXTI;
END;
GOTO FOUND;
NEXTI:;
END;
END;
I,NAMELI=NAMELI+1;
IF I>1256 THEN
DO ;
CALL NEWLIN;
PUT FILE(SYSPRINT)EDIT('TOO MANY NAMES')(A);
GOTO ERROR;
END;
DO J=1,2,3;
NAMETA(I,J)=BUFF(J);
END;
GOTO FOUND;
L1:DO I=($CONAEX(J=4,I+1,I)) BY 1 TO 3;
DO J=($CONAEX(J=4,1,J+1)) BY 1 TO 4;
BUFF(I)=256*BUFF(I)+63;
END;
```

(SUBRG) :

```
END;
SAVE=A;
GOTO LOOK;
FOUND:A=I;
EXIT:$RTURN=A;
        IF A = LCOMMENT | A = RCOMMENT | A = SCOMMENT THEN DO;
        CALL MESSAGE('ILLEGAL COMMENT SYMBOL @ % OR & IS IGNORED.');
        GOTO STARTA; END;
IF OUTPRINT THEN DO; UNIT=OUTUNI; OUTUNI=PRINT;
  IF A < 0 THEN DO; CALL OUTPUT(63);
      IF A=LEFTBRACE THEN DO;
              IF NAMECOUNT=6 THEN DO;
              NAMECOUNT=0;
              PUT EDIT(' ')(SKIP(2),COLUMN(21),A);END;
              CALL OUTPUT(49);END;
     ELSE IF A=RIGHTBRACE THEN CALL OUTPUT(50);
          ELSE IF A=LBRACK THEN DO;
                  IF NAMECOUNT=6 THEN DO;
                  NAMECOUNT=0;
                  PUT EDIT(' ')(SKIP(2),COLUMN(21),A);END;
                  CALL OUTPUT(61);END;
               ELSE IF A=RBRACK THEN CALL OUTPUT(62);
                    ELSE IF A=ONEOUT THEN CALL OUTPUT(58);
                         ELSE IF A=THREEOUTS THEN CALL OUTPUT(56);
                            ELSE IF A=ENDALT THEN CALL OUTPUT(59);
                END;
  ELSE IF FIRSTC -= 0 THEN DO;
              IF NAMECOUNT=6 THEN DO;
              NAMECOUNT=0;
              PUT EDIT(' ')(SKIP(2),COLUMN(21),A);
              END;
              CALL PRINTN(A);
              NAMECOUNT=NAMECOUNT+1;END;
  OUTUNI=UNIT;END;
ELSE OUTPRINT='1'B;
END/*FINDNAME */;
RETURN($RTURN);
END;
        DCL DEFINITION ENTRY ;
DEFINITION: PROC RECURSIVE ;
        BEGIN ; DCL ID FIXED BINARY(31) ; ID=0 ;
        CALL SEQUENCE ;
        ID=FINDNA ;
        IF ID=ENDALT THEN
        DO ;
        TABLIST1(CRULE)=AP(TABLIST1(CRULE),TABLIST2(CALT)) ;
        TABLIST2(CALT)=0 ;
```

```
            CALL DEFINITION ;
            END ;
            ELSE CNAME=ID ;
            END ;
END DEFINITION ;
            DCL SEQUENCE ENTRY ;
  SEQUENCE: PROC RECURSIVE ;
            BEGIN ; DCL IS FIXED BINARY(31) ; IS=0 ;
            CALL UNIT ;
            IS=FINDNA ;
            IF IS=NEWRUL | IS=ENDALT | IS=RIGHTBRACE | IS=RBRACK THEN
            CNAME=IS ;
            ELSE
            DO ;
            CNAME=IS ;
            CALL SEQUENCE ;
            END ;
            END ;
END SEQUENCE ;
            DCL UNIT ENTRY ;
UNIT: PROC RECURSIVE ;
            BEGIN ; DCL(IU,SNAME,SEPARATOR) FIXED BINARY(31) ;
            SNAME=0;
            IF BACKLBRACK < 2 THEN
            BACKLBRACK=BACKLBRACK+1;
            IU=FINDNA ;
            IF IU > 0 THEN
            DO ;
            TABLIST2(CALT)=AP(TABLIST2(CALT),IU) ;
            SNAME=IU ;
            IU=FINDNA ;
            IF IU=THREEDOTS THEN
            DO ;
            IF SNAME < FIRSTB THEN
            CALL MESSAGE('METASYMBOL + AFTER FUNCTION NAME');
            ELSE
            DO;
            IU=FINDNA;
            IF IU ¬= RBRACK | BACKLBRACK ¬= 1 THEN
            DO;
            NEWCLASS=NEWCLASS+1 ;
            WLIST=CREATERULE(NEWCLASS,SNAME,NEWCLASS);
            CALL OUTLIS((WLIST)) ;
            END;
            TABLIST2(CALT) = AP(TABLIST2(CALT),NEWCLASS);
            CNAME=IU;
            END;
```

(SUBRG) :

```
            END ;
            ELSE CNAME=IU ;
/*  FIN DU TRAITEMENT DE UNIT ::=
                    BASICNAME | CLASSNAME | BASICNAME + | CLASSNAME +
            END ;
            ELSE
            IF IU=LEFTBRACE THEN
            DO ;
            NEWCLASS=NEWCLASS+1 ;
            CRULE=CRULE+1 ;
            CALT=CALT+1 ;
            TABLIST1(CRULE)=CONS(NEWCLASS,0) ;
            TABLIST2(CALT)=0 ;
            TABCLASS(CRULE)=NEWCLASS ;
            IU=FINDNA ;
            IF IU ¬= ONEDOT THEN
            DO ;
            CNAME=IU ;
            CALL DEFINITION ;
            IU=FINDNA ;
            IF IU=RIGHTBRACE THEN
            DO ;
            TABLIST1(CRULE)=AP(TABLIST1(CRULE),TABLIST2(CALT)) ;
            CALL OUTLIS((TABLIST1(CRULE))) ;
            TABLIST2(CALT)=0 ;
            CALT=CALT-1 ;
            TABLIST2(CALT) = AP(TABLIST2(CALT),TABCLASS(CRULE));
            IU=FINDNA ;
            IF IU=THREEDOTS THEN
            DO ;
            NEWCLASS=NEWCLASS+1 ;
            TABLIST2(CALT)=AP(TABLIST2(CALT),NEWCLASS) ;
            WLIST=CREATERULE(NEWCLASS,TABCLASS(CRULE),NEWCLASS);
            CALL OUTLIS((WLIST));
            END ;
            ELSE CNAME=IU ;
            TABLIST1(CRULE)=0;
            TABCLASS(CRULE)=0;
            CRULE = CRULE - 1;
            END ;
            ELSE DO;
            CALL MESSAGE('EXPECTED METASYMBOL > IS NOT FOUND');
            CNAME=IU;END;
/* FIN DU TRAITEMENT DE UNIT ::=
                                < DEFINITION > | < DEFINITION > +
            END ;
            ELSE
```

```
DO ;
SEPARATOR=FINDNA ;
IF SEPARATOR < FIRSTB | SEPARATOR >= FIRSTC THEN
CALL MESSAGE('SYMBOL USED AS SEPARATOR IS NOT A BASIC SYMBOL'
IU=FINDNA ;
IF IU ¬= ONEDOT THEN
DO ;
CALL MESSAGE('METASYMBOL . OMITTED AFTER SEPARATOR');
CNAME=IU;
END ;
CALL UNIT ;
IU=FINDNA ;
IF IU ¬= ONEDOT THEN
DO ;
CALL MESSAGE('METASYMBOL . OMITTED AFTER UNIT');
CNAME=IU;
END;
IU=FINDNA ;
IF IU ¬= RIGHTBRACE THEN
DO ;
CALL MESSAGE('METASYMBOL > OMITTED AFTER METASYMBOL . ');
CNAME=IU;
END ;
ELSE
DO ;
STACK= AP(TABLIST2(CALT),TABCLASS(CRULE));
TABLIST2(CALT)=CONS(SEPARATOR,STACK);
TABLIST1(CRULE) = AP(TABLIST1(CRULE),0);
TABLIST1(CRULE) = AP(TABLIST1(CRULE),TABLIST2(CALT));
CALL OUTLIS((TABLIST1(CRULE)));
TABLIST1(CRULE)=0;
TABCLASS(CRULE)=0;
CALT = CALT - 1;
CRULE = CRULE - 1;
DO WHILE( STACK ¬= 0 );
ITEM=HD(STACK);
TABLIST2(CALT) = AP(TABLIST2(CALT),ITEM);
STACK=TL(STACK);
END;
END ;
/*   FIN DU TRAITEMENT DE UNIT ::= <. SEPARATOR . UNIT .>
END ;
END ;
ELSE
IF IU=LBRACK THEN
DO ;
NEWCLASS=NEWCLASS+1 ;
```

```
CRULE=CRULE+1 ;
CALT=CALT+1 ;
TABLIST1(CRULE)=CONS(NEWCLASS,0) ;
TABLIST2(CALT)=0 ;
TABCLASS(CRULE)=NEWCLASS ;
IU=FINDNA ;
IF IU ¬= ONEDOT THEN
DO ;
BACKLBRACK = 0 ;
CNAME=IU ;
CALL DEFINITION ;
IU=FINDNA ;
IF IU=RBRACK THEN
DO ;
TABLIST1(CRULE)=AP(TABLIST1(CRULE),TABLIST2(CALT));
IU=FINDNA ;
IF IU ¬= THREEDOTS THEN
DO ;
CNAME=IU ;
TABLIST1(CRULE)=AP(TABLIST1(CRULE),0) ;
CALL OUTLIS((TABLIST1(CRULE))) ;
TABLIST2(CALT)=0 ;
CALT=CALT-1 ;
TABLIST2(CALT)=AP(TABLIST2(CALT),TABCLASS(CRULE)) ;
CRULE=CRULE-1 ;
END ;
ELSE
DO ; CALL OUTLIS((TABLIST1(CRULE))) ;
NEWCLASS=NEWCLASS+1 ;
WLIST=CREATERULE(NEWCLASS,TABCLASS(CRULE),NEWCLASS);
CALL OUTLIS((WLIST));
TABCLASS(CRULE)=0 ;
CRULE=CRULE-1 ;
CALT=CALT-1 ;
TABLIST2(CALT)=AP(TABLIST2(CALT),NEWCLASS) ;
END ;
END ;
ELSE DO;
CALL MESSAGE('EXPECTED METASYMBOL ) IS NOT FOUND');
CNAME=IU;END;
/*   FIN DU TAITEMENT DE UNIT ::= ( DEFINITION ) | ( DEFINITION ) +
END ;
ELSE
DO;
SEPARATOR=FINDNA;
IF SEPARATOR < FIRSTB | SEPARATOR >= FIRSTC THEN
CALL MESSAGE('SYMBOL USED AS SEPARATOR IS NOT A BASIC SYMBOL'
```

```
            IU=FINDNA;
            IF IU ¬= ONEDOT THEN
            DO;
            CALL MESSAGE('METASYMBOL . OMITTED AFTER SEPARATOR');
            CNAME=IU;
            END;
            CALL UNIT;
            IU=FINDNA;
            IF IU ¬= ONEDOT THEN
            DO;
            CALL MESSAGE('METASYMBOL . OMITTED AFTER UNIT');
            CNAME=IU;
            END;
            IU=FINDNA;
            IF IU ¬= RBRACK THEN
            DO;
            CALL MESSAGE('METASYMBOL ) OMITTED AFTER METASYMBOL . ');
            CNAME=IU;
            END;
            ELSE
            DO;
            NEWCLASS=NEWCLASS+1 ;
            TABLIST2(CALT)=AP(TABLIST2(CALT),NEWCLASS) ;
            TABLIST1(CRULE)=AP(TABLIST1(CRULE),0) ;
            TABLIST1(CRULE)=AP(TABLIST1(CRULE),TABLIST2(CALT)) ;
            CALL OUTLIS((TABLIST1(CRULE))) ;
            STACK=CONS(SEPARATOR,TABLIST2(CALT)) ;
            WLIST=CONS(NEWCLASS,0) ;
            WLIST=AP(WLIST,0) ;
            WLIST=AP(WLIST,STACK) ;
            CALL OUTLIS((WLIST)) ;
            TABLIST1(CRULE)=0 ;
            CALT=CALT-1 ;
            TABLIST2(CALT)=AP(TABLIST2(CALT),TABCLASS(CRULE)) ;
            CRULE=CRULE-1 ;
            END ;
/* FIN DU TRAITEMENT DE UNIT ::= (.SEPARATOR.UNIT.) */
            END ;
            END;
            ELSE
            IF IU=THREEDOTS | IU=ONEDOT THEN
            CALL MESSAGE('ILLEGAL METASYMBOL + OR .');
            ELSE IF IU=ENDALT | IU=NEWRUL | IU=RIGHTBRACE | IU=RBRACK
                  THEN CNAME=IU;
END UNIT ;
            DCL MESSAGE ENTRY( );
MESSAGE: PROC(A) ;
```

```
          DCL A CHAR(*);
          CALL NEWLIN; CALL NEWLIN;
          PUT FILE(SYSPRINT)EDIT('/// '||A)(A);
          PUT FILE(SYSPRINT) EDIT(' ')(SKIP(2),COLUMN(20),A);
          NAMECOUNT=0;
          CONTIN='0'B;
END MESSAGE ;
          DCL CREATERULE ENTRY(FIXED BIN(31),FIXED BIN(31),FIXED BIN(31)
          RETURNS(FIXED BIN(31)) ;
CREATERULE: PROC(X,Y,Z) FIXED BIN(31) ;
          DCL(X,Y,Z,$RTURN) FIXED BIN(31) ;
          ALTLIS = CONS(Y,CONS(Z,0));
          RHSLIS = CONS(0,CONS(ALTLIS,0));
        $RTURN = CONS(X,RHSLIS);
          RETURN($RTURN) ;
END CREATERULE ;
CALL NEWPAG;
OUTUNI=DISKA;
SAVE=0;
NAMELI=0;
STARTB=-1;
STARTF=-2;
STARTR=-3;
ENDRUL=-4;
ENDALT=-5;
NEWRUL=-6;
LBRACK=-7;
RBRACK=-8;
          LEFTBRACE=-9 ;
          RIGHTBRACE=-10 ;
          ONEDOT=-11 ;
          THREEDOTS=-12 ;
          LCOMMENT=-13;
          RCOMMENT=-14;
          SCOMMENT=-15;
          BACKLBRACK=2;
          LISTARRAY='1'B;
          OUTPRINT='0'B;; FIRSTC=0;
          GET FILE(SYSIN)LIST(LASTCLASS);
          NEWCLASS=LASTCLASS;
          CNAME=0;
          REFSIZ=0;
          NAMECOUNT=0;
          DO J=1 BY 1 TO 400 ;
          CODE(J)=0;
          END;
I=FINDNA;
```

(SUBRG) :

```
IF I¬=STARTF THEN
DO ;
PUT FILE(SYSPRINT)EDIT('NO 1 FOUND AT FRONT OF DECK')(A);
GOTO ERROR;
END;
        INDEX$= 0;
        $LCP4: INDEX$= INDEX$+1;
DO ;
REDO:I=FINDNA;
IF I=STARTB THEN
DO ;
FIRSTB=INDEX$;
PUT FILE(SYSPRINT)EDIT('FIRSTBASIC=')(A);
CALL ITEMPR((FIRSTB));
CALL NEWLIN;
GOTO REDO;
END;
IF I=STARTR THEN
GOTO READRU;
IF I¬=INDEX$ THEN
DO ;
CALL NEWLIN;
INDEX$=INDEX$-1;
IF I<FIRSTB THEN
PUT FILE(SYSPRINT)EDIT('FUNCTION')(A);
ELSE PUT FILE(SYSPRINT)EDIT('BASIC SYMBOL')(A);
CALL ITEMPR((I));
CALL PRINTN((I));
PUT FILE(SYSPRINT)EDIT(' REPEATED AFTER')(A);
CALL ITEMPR((INDEX$));
CALL PRINTN((INDEX$));
CALL NEWLIN;
CONTIN='0'B;
END;
END;
        GOTO  $LCP4;
READRU:FIRSTC=INDEX$;
PUT FILE(SYSPRINT)EDIT('FIRSTCLASS=')(A);
CALL ITEMPR((FIRSTC));
INDEX$=1; CALL NEWPAG;
CALL PRINTI('INPUT SYNTAX='); OUTPRINT='1'B;
NAMECOUNT = -1;
CALL ITEMPR((INDEX$)); PUT EDIT(' ')(COLUMN(5),A);
        PROD_RULE :
DO ;
CRULE=1 ;
CALT=1 ;
```

```
UBRG) :

        CALL DCLISTARRAY(ZERO);
        DO J=1 BY 1 TO 100 ;
        TABCLASS(J)=0 ;
        END ;
        NEWNAM : I = FINDNA;
        IF I<0 THEN
        DO ;
        PUT EDIT('/// ILLEGAL USE AFTER METASYMBOL 3 OR * OF ONE OF METASYMBOL
         1 2 3 , ( ) < > . + * WAS IGNORED')(SKIP(2),A);
        CALL NEWLIN;CALL NEWLIN;
        GOTO NEWNAM;
        END;
        ELSE IF I < FIRSTC THEN
          DO;CALL NEWLIN;
          IF I < FIRSTB THEN
          PUT FILE(SYSPRINT)EDIT('/// FUNCTION')(SKIP(2),A);
          ELSE PUT FILE(SYSPRINT)EDIT('/// BASIC SYMBOL')(SKIP(2),A);
          END;
             ELSE DO;
                IF INDEX$ > 400 THEN DO;
                PUT EDIT('/// TOO MANY CLASSNAMES')(SKIP(2),A);
                GOTO ERROR;END;
                CODE(INDEX$) = I;
                IF INDEX$ > 1 THEN
             DO J = 1 BY 1 TO INDEX$ - 1;
             IF CODE(J) = I THEN
             DO; CALL NEWLIN;
             PUT EDIT('/// PREVIOUSLY DEFINED CLASSNAME')(SKIP(2),A);
             GOTO LHS;
             END;
                END;
                GOTO RIGHTHANDSIDE;END;
         LHS : CALL ITEMPR((I));
                CALL PRINTN((I)); CALL NEWLIN;
                CALL MESSAGE('APPEARS AS LEFT HAND SIDE');
                INDEX$=INDEX$-1;
         RIGHTHANDSIDE : PUT EDIT(' ')(COLUMN(20),A);
         TABLIST1(CRULE) = CONS(I,0);
         CALL DEFINITION ;
         IP=FINDNA;
         IF  IP=NEWRUL  THEN
                DO ;
                TABLIST1(CRULE)=AP(TABLIST1(CRULE),TABLIST2(CALT)) ;
                CALL OUTLIS((TABLIST1(CRULE))) ;
                END ;
                ELSE  DO;
                  CALL MESSAGE('ILLEGAL METASYMBOL > OR )');
```

(SUBRG) :

```
            GOTO RIGHTHANDSIDE; END;
      CALL NEWLIN;CALL NEWLIN;
      CALL ITEMPR((INDEX$+1));PUT EDIT(' ')(COLUMN(5),A);
          NAMECOUNT = -1;
          IP=FINDNA;
          IF IP = ENDRUL THEN DO;
          PUT FILE(SYSPRINT)EDIT(' /// END OF OLD GRAMMAR')(A);
          CALL NEWLIN;
          GOTO DONE; END;
          ELSE  CNAME=IP;
          INDEX$=INDEX$+1 ;
          GOTO PROD_RULE;
END /* PROD_RULE */;
DONE: CALL OUTLIS(0) ;
LASTCL=FIRSTC+INDEX$-1;
IF NAMELI ¬= LASTCL THEN
DO; CONTIN='0'B;
DO I=FIRSTC BY 1 TO NAMELI;
  DO;
  DO J=1 BY 1 TO INDEX$;
  IF CODE(J) = I THEN
  GOTO FINDCODE;
  END;
  CALL NEWLIN;
  CALL ITEMPR((I));
  CALL PRINTN((I));
  PUT FILE(SYSPRINT) EDIT (' IS UNDEFINED CLASSNAME')(A);
  CALL NEWLIN;
FINDCODE:
  END;
END;
END;
LASTCL=NAMELI;
          CALL NEWLIN;
          PUT FILE(SYSPRINT) EDIT('OLD GRAMMAR:LAST CLASSNAME=')(A)
          CALL ITEMPR((LASTCL)) ;
          PUT EDIT('BNF GRAMMAR : LAST CLASSNAME=')(SKIP(2),A);
          CALL ITEMPR((NEWCLASS)) ;
          CALL NEWLIN ;
          IF LASTCLASS¬=LASTCL THEN DO;
PUT EDIT('LAST CLASSNAME IS FALSE IN THE FOURTH DATUM')(SKIP(2),A)
          CONTIN='0'B;END;
          IF ¬(CONTIN) THEN GOTO ERROR;
          LISTARRAY='0' ;
          REFSIZ=FLOOR((NEWCLASS-1)/(1E0*WORDLE))+1 ;
BNF_SYNTAX : BEGIN;
          DCL(REF(FIRSTC:NEWCLASS,1:REFSIZ)) FIXED BIN(31) ;
```

```
          DCL INDEXT(FIRSTC:LASTCL) FIXED BIN(31);
          DCL(CLASS,ITEM,WORD,BIT,ITEMCN) FIXED BIN(31) ;
          DCL(INFUNC,HADITE) BIT(1) ;
          DO CLASS = FIRSTC BY 1 TO NEWCLASS;
          DO WORD=1 BY 1 TO REFSIZ ;
          REF(CLASS,WORD)=0 ;
          END ;
          END ;
          DO I = FIRSTC BY 1 TO LASTCL;INDEXT(I) = 0; END;
          DO CLASS = FIRSTC BY 1 TO LASTCL;
                DO I = 1 BY 1 TO (LASTCL-FIRSTC+1);
                IF INDEXT(CLASS) = 0 THEN
                     IF CODE(I) = CLASS THEN
                     INDEXT(CLASS) = I;
                END;
          END;
          OUTUNI=PRINT ;
          INUNIT=DISKA;
          CALL BACKSP((DISKA));
          CALL NEWPAG;
          CALL PRINTI('BNF SYNTAX =');
READRULE: RULE=INLIST ;
          IF RULE ¬= 0 THEN
          DO ;
          CLASS=HD(RULE) ;
          IF CLASS <= LASTCL THEN DO;
                CALL OUTITE((INDEXT(CLASS)));
                PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(8),A);
                CALL OUTITE((CLASS)); END;
          PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(16),A);
          IF CLASS <= LASTCL THEN CALL PRINTN((CLASS)) ;
          ELSE CALL OUTITE((CLASS)) ;
READALT: RULE=TL(RULE) ;
          IF RULE ¬= 0 THEN
          DO ;
          ALT=HD(RULE) ;
          PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(31),A);
          IF ALT=0 THEN
          DO ;
          CALL OUTPUT(63) ;
          CALL OUTPUT(61) ;
          CALL OUTPUT(62) ;
          END ;
NEXTITEM: IF ALT ¬= 0 THEN
          DO ;
          ITEM=HD(ALT) ;
          IF ITEM <= LASTCL THEN CALL PRINTN((ITEM)) ;
```

```
                ELSE CALL OUTITE((ITEM)) ;
                IF CLASS >= FIRSTC THEN
                DO ;
                WORD=(ITEM-1)/WORDLE+1 ;
                BIT=ITEM-WORDLE*(WORD-1) ;
                REF(CLASS,WORD)=REF(CLASS,WORD)|BITMAS(BIT) ;
                END ;
                ALT=TL(ALT) ;
                GOTO NEXTITEM ;
                END ;
                CALL NEWLIN ;
                GOTO READALT ;
                END ;
                CALL NEWLIN ;
                GOTO READRULE ;
                END ;
        CALL DECLAR(ZERO) ;
        CALL NEWPAG ;
                CALL PRINTI('CROSS REFERENCE TABLE*FUNCTIONS') ;
                INFUNC='1'B ;
                DO ITEM = 1 BY 1 TO NEWCLASS;
                DO ;
                WORD=(ITEM-1)/WORDLE+1 ;
                BIT=ITEM-WORDLE*(WORD-1) ;
                IF ITEM=FIRSTB THEN
                DO ;
                CALL NEWPAG ;
                INFUNC='0'B ;
                CALL PRINTI('CROSS REFERENCE TABLE*BASIC SYMBOLS') ;
                END ;
                IF ITEM=FIRSTC THEN
                DO ;
                CALL NEWPAG ;
                CALL PRINTI('CROSS REFERENCE TABLE*CLASSNAMES') ;
                END ;
                IF ¬ (INFUNC) THEN
                DO ;
                IF ITEM <= LASTCL THEN DO;
                      CALL OUTTHI((ITEM));
                      IF ITEM >= FIRSTC THEN DO;
                           CALL OUTPUT(63);CALL OUTPUT(61);
                           IF INDEXT(ITEM) ¬= 0 THEN
                           CALL OUTTHI((INDEXT(ITEM)));
                           ELSE CALL OUTPUT(10);
                           CALL OUTPUT(62); END; END;
               PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(11),A);
               IF ITEM <= LASTCL THEN CALL PRINTN((ITEM)) ;
```

```
            ELSE CALL OUTITE((ITEM)) ;
            PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(27),A);
            END ;
            ITEMCN=0 ;
            HADITE='0'B ;
            DO CLASS = FIRSTC BY 1 TO NEWCLASS;
            DO ;
            IF MOD(REF(CLASS,WORD),BITMAS(BIT+1)) >= BITMAS(BIT) THEN
            DO ;
            IF INFUNC & ¬ (HADITE) THEN
            DO ;
            HADITE='1'B ;
            IF ITEM <= LASTCL THEN DO;
                  CALL OUTTHI((ITEM));
                  IF ITEM >= FIRSTC THEN DO;
                        CALL OUTPUT(63);CALL OUTPUT(61);
                        IF INDEXT(ITEM) ¬= 0 THEN
                        CALL OUTTHI((INDEXT(ITEM)));
                        ELSE CALL OUTPUT(10);
                        CALL OUTPUT(62); END; END;
            PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(11),A);
            IF ITEM <= LASTCL THEN CALL PRINTN((ITEM)) ;
            ELSE CALL OUTITE((ITEM)) ;
            PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(27),A);
            END ;
            CALL OUTITE((CLASS)) ;
            IF CLASS <= LASTCL THEN DO;
            CALL OUTPUT(61);
            IF INDEXT(CLASS) ¬= 0 THEN
            CALL OUTTHI((INDEXT(CLASS)));
            ELSE CALL OUTPUT(10);
            CALL OUTPUT(62);
            CALL OUTPUT(63);END;
            ITEMCN=ITEMCN+1 ;
            IF ITEMCN = 8 THEN
            DO ;
            ITEMCN=0 ;
            CALL NEWLIN ;
            PUT FILE(SYSPRINT)EDIT(' ')(COLUMN(27),A);
            END ;
            END ;
            END ;
            END ;
            IF ¬ (INFUNC) | HADITE THEN
            DO ;
            CALL NEWLIN ;
            CALL NEWLIN ;
            END ;
            END ;
            END ;
      END BNF_SYNTAX;
            LASTCL=NEWCLASS;
      END INPUTGRAMMAR;
```

CHAPITRE VI

APPLICATIONS

CHAPITRE VI

APPLICATIONS

6. APPLICATIONS

6.1. Programme REC (élimination de la récursivité à gauche)

Ce programme a été utilisé pour tester une version d'une grammaire context-free d'ALGOL 68 du M.B.L.E. (la dernière se trouve en [Bra]), grammaire dont s'est servi M. Simonet au début de son travail sur ALGOL 68 [Sim].

Elle comportait 29 symboles non-terminaux récursifs à gauche. On a trouvé 27 cliques maximales :

26 contenant un symbole non-terminal

1 contenant 3 symboles non-terminaux

Le nombre des récursivités à gauche s'explique par le fait que cette grammaire est prévue pour un analyseur ascendant à contexte borné [Loe]. On sait qu'en analyse descendante la récursivité à gauche n'est pas gênante.

L'algorithme d'élimination, tel qu'il est présenté dans § 2.2.3, calcule chaque clique maximale puis transforme les règles correspondant à cette clique. Avant d'effectuer cette transformation, le programme balaye la grammaire pour chercher ces règles. Mais cette méthode oblige à de nombreux balayages. Nous avons diminué ces balayages en utilisant le fait que les cliques maximales de plus d'un élément sont rares.

La grammaire est écrite en deux exemplaires sur deux fichiers différents. Un fichier est lu séquentiellement (cliques à un seul élément), l'autre est initialisé à chaque recherche de règles correspondantes à une clique maximale ayant plus d'un élément.

Avec les programmes BNF + REC on a obtenu (temps unité centrale) 3 minutes avec la première méthode, 2 minutes 7" avec la méthode optimisée. (La grammaire a 71 symboles terminaux et 116 non-terminaux).

Mais avec une grammaire issue du programme ULD, la lecture séquentielle des règles de la grammaire n'est plus possible car les numéros des symboles non-terminaux n'ont pas le même ordre que les règles de la grammaire. Le programme doit être modifié de la manière suivante : (cf. algorithme § 2.2.3).

Le Pas 4 est remplacé par :

Pas 4 : on crée une liste composée des éléments de C_i et on l'ajoute comme sous-liste à une liste spéciale : splist. Ces éléments sont rangés dans un tableau sparray. A la suite du Pas 5 viennent les Pas 6 et Pas 7.

Pas 6 : on recopie sur un fichier spfile les règles correspondantes aux symboles non-terminaux figurant dans sparray (symboles récursifs à gauche).

Pas 7 : pour chaque sous-liste de splist, appel de la procédure de transformation avec, en paramètre, les symboles figurant dans la sous-liste ainsi que leur nombre; la procédure traite les règles figurant dans spfile correspondantes aux symboles non-terminaux fournis, calcule les matrices A et B (cf. § 2.2.2 et 2.2.3), crée les nouveaux symboles et ces nouvelles règles. On recommence le Pas 7 tant que la liste splist n'est pas vide.

A la différence du premier, ce nouveau programme calcule toutes les cliques maximales dans une première étape, ensuite dans une deuxième étape les règles correspondantes à ces cliques sont successivement transformées.

Remarquons que cette méthode est toujours meilleure si le nombre des symboles récursifs à gauche est petit devant le nombre total de symboles et/ou si le nombre de cliques ayant plus d'un élément n'est pas petit devant le nombre total de cliques.

Ce n'est d'ailleurs pas le cas dans la grammaire d'ALGOL 68 que l'on a testée.

INPUT SYNTAX=

```
72      PROGRAM        LABSEQ OPEN SERCL CLOSE
                       OPEN SERCL CLOSE

73      OPEN           'LEFTPAR'
                       'BEGIN'

74      CLOSE          'RIGHTPAR'
                       'END'

75      LABSEQ         LAB
                       LABSEQ LAB

76      LAB            'IDEN' 'TWOPOINTS'

77      SERCL          DECPREF LABOPUNI
                       SERIES1 LABOPUNI
                       LABOPUNI

78      DECPREF        DEC 'SEMICOLON'
                       UNICL 'SEMICOLON'
                       DECPREF DEC 'SEMICOLON'
                       DECPREF UNICL 'SEMICOLON'

79      SERIES1        DECPREF LABUNI 'SEMICOLON'
                       DECPREF LABOPUNI COMPL
                       LABUNI 'SEMICOLON'
                       LABOPUNI COMPL
                       SERIES1 LABOPUNI 'SEMICOLON'
                       SERIES1 LABOPUNI COMPL

80      LABOPUNI       LABUNI
                       UNICL

81      LABUNI         LABSEQ UNICL

82      COMPL          'POINT' LAB

83      DEC            UNIDEC
                       DEC 'COMMA' UNIDEC

84      UNIDEC         MODEC
                       EXMODEC
                       STRDEC
                       EXSTRDEC
                       UNODEC
                       EXUNODEC
                       PRIDEC
                       EXPRIDEC
                       IDEDEC
                       EXIDEDEC
                       OPDEC
                       EXOPDEC

85      MODEC          'MODE' 'MODIND' 'EQUAL' DECER
```

86	STRDEC	'STRUCT' 'MODIND' 'EQUAL' FOPPK
87	UNODEC	'UNION' 'MODIND' 'EQUAL' DERPK
88	EXMODEC	MODEC 'COMMA' 'MODIND' 'EQUAL' DECER EXMODEC 'COMMA' 'MODIND' 'EQUAL' DECER
89	EXSTRDEC	STRDEC 'COMMA' 'MODIND' 'EQUAL' FOPPK EXSTRDEC 'COMMA' 'MODIND' 'EQUAL' FOPPK
90	EXUNODEC	UNODEC 'COMMA' 'MODIND' 'EQUAL' DERPK EXUNODEC 'COMMA' 'MODIND' 'EQUAL' DERPK
91	PRIDEC	'PRIOR' 'OPIND' 'EQUAL' 'NUMB'
92	EXPRIDEC	PRIDEC 'COMMA' 'OPIND' 'EQUAL' 'NUMB' EXPRIDEC 'COMMA' 'OPIND' 'EQUAL' 'NUMB'
93	IDEDEC	IDEDE1 IDEDE2
94	IDEDE1	DECER IDEQACPAR
95	IDEDE2	DECER IDINITOP
96	IDEQACPAR	'IDEN' 'EQUAL' ACPAR
97	IDINITOP	'IDEN' 'IDEN' 'BECOMES' UNICL
98	EXIDEDEC	EXIDEDE1 EXIDEDE2 STRDE1 STRDE2 UNODE1 UNODE2
99	EXIDEDE1	IDEDE1 'COMMA' IDEQACPAR EXIDEDE1 'COMMA' IDEQACPAR
100	EXIDEDE2	IDEDE2 'COMMA' IDINITOP EXIDEDE2 'COMMA' IDINITOP
101	STRDE1	STRDECOR IDEQACPAR STRDE1 'COMMA' FOIDEQACP
102	STRDE2	STRDECOR IDINITOP STRDE2 'COMMA' FOIDINIT
103	UNODE1	UNODECOR IDEQACPAR UNODE1 'COMMA' VOIDEQACP
104	UNODE2	UNODECOR IDINITOP UNODE2 'COMMA' VOIDINIT
105	FOIDEQACP	FOPPK IDEQACPAR
106	FOIDINIT	FOPPK IDINITOP

```
107     VOIDEQACP       DERPK IDEQACPAR

108     VOIDINIT        DERPK IDINITOP

109     FOPPK           'LEFTPAR' FOPARS 'RIGHTPAR'

110     FOPARS          FPARBASE
                        FOPARS 'COMMA' FPARBASE

111     FPARBASE        FPARB1
                        FPARSTR
                        FPARUNO

112     FPARB1          DECER 'IDEN'
                        FPARB1 'COMMA' 'IDEN'

113     FPARSTR         'STRUCT' FOPPK 'IDEN' 'COMMA' FOPPK 'IDEN'
                        FPARSTR 'COMMA' FOPPK 'IDEN'

114     FPARUNO         'UNION' DERPK 'IDEN' 'COMMA' DERPK 'IDEN'
                        FPARUNO 'COMMA' DERPK 'IDEN'

115     DERPK           'LEFTPAR' DECERS 'RIGHTPAR'

116     DECERS          DECER
                        DECERS 'COMMA' DECER

117     OPDEC           'OP' OPBASE
                        OPDEC 'COMMA' OPBASE

118     EXOPDEC         EXOPDEC 'COMMA' OPBASE
                        OPDEC 'COMMA' OPBASE

119     OPBASE          VIRPLAN OPEQACPAR
                        'OPIND' 'EQUAL' RTNDEN

120     OPEQACPAR       'OPIND' 'EQUAL' ACPAR

121     DECER           DECOR
                        'MODIND'

122     DECOR           DECO1
                        DECO2

123     DECO1           PRIDECOR
                        LONDECOR
                        REFDECOR
                        ROWDECOR
                        PRODECOR
                        'COMPLEX'
                        'BITS'

124     PRIDECOR        'INT'
                        'REAL'
                        'BOOL'
                        'CHAR'
                        'FORMAT'
```

125	LONDECOR	LONSEQ 'REAL' LONSEQ 'INT' LONSEQ 'COMPLEX' LONSEQ 'BITS'
126	LONSEQ	'LONG' LONSEQ 'LONG'
127	ROWDECOR	'LEFTPAR' ROWER 'RIGHTPAR' DECER 'LEFTBRACK' ROWER 'RIGHTBRACK' DECER 'LEFTPAR' 'RIGHTPAR' DECER 'LEFTBRACK' 'RIGHTBRACK' DECER
128	ROWER	'COMMA' ROWER 'COMMA' BOUNDS ROWER 'COMMA'
129	BOUNDS	'TWOPOINTS' BOUND 'TWOPOINTS' BOUND BOUND 'TWOPOINTS' 'TWOPOINTS' BOUND
130	BOUND	UNICL UNICL 'FLEX' 'FLEX' 'EITHER' UNICL 'EITHER'
131	REFDECOR	'REF' DECER
132	PRODECOR	'PROC' 'PROC' VIRPLAN
133	VIRPLAN	DERPK DECER DERPK DECER
134	DECO2	STRDECOR UNODECOR
135	STRDECOR	'STRUCT' FOPPK
136	UNODECOR	'UNION' DERPK
137	UNICL	TERT CONFR
138	TERT	SECOND FORMUL FORCL
139	SECOND	PRIM COHES
140	PRIM	CLSTARCL BASE

141	CONFR	NLASSIG CONFREL IDENREL
142	NLASSIG	'NLDEST' 'BECOMES' SOURCE
143	NLPEST	TERT
144	SOURCE	UNICL
145	LASSIG	LDEST 'BECOMES' SOURCE
146	LDEST	LGEN
147	CONFREL	TERT CONRELOR TERT
148	IDENREL	TERT IDERELOR TERT
149	CONRELOR	'CONFORMSTO' 'CONFANDBEC'
150	IDERELOR	'ISSYMBOL' 'ISNOTSYMBOL'
151	FORMUL	OPERAND 'OPIND' OPERAND FORMUL 'OPIND' OPERAND
152	OPERAND	SECOND 'OPIND' SECOND
153	COHES	NLGEN SELECT
154	NLGEN	DECER
155	LGEN	'LOC' DECER
156	SELECT	'IDEN' 'OF' SECOND
157	BASE	SLICALL DENOT 'IDEN'
158	SLICALL	PRIM SUB INDEX BUS PRIM SUB BUS
159	SUB	'LEFTPAR' 'LEFTBRACK'
160	BUS	'RIGHTPAR' 'RIGHTBRACK'
161	INDEX	SUBSCR TRIM 'COMMA' INDEX 'COMMA' INDEX 'COMMA' SUBSCR INDEX 'COMMA' TRIM

```
COLLECT GARBAGE
 162      SUBSCR          ACPAR

 163      TRIM            UNITWOPUNI 'ATUNI'
                          UNITWOPUNI

 164      UNITWOPUNI      UNICL 'TWOPOINTS'
                          'TWOPOINTS' UNICL
                          'TWOPOINTS'

 165      ACPAR           UNICL
                          LGEN
                          LASSIG

 166      DENOT           'INTDEN'
                          'LINTDEN'
                          'REALDEN'
                          'LREALDEN'
                          'BOOLDEN'
                          'CHARDEN'
                          'BITSDEN'
                          'STRIDEN'
                          RTNDEN
                          'FORMDEN'

 167      RTNDEN          'LEFTPAR' FOPLAN RTNSYM BODY 'RIGHTPAR'
                          'LEFTPAR' RTNSYM BODY 'RIGHTPAR'

 168      FOPLAN          FOPPK DECER
                          FOPPK
                          DECER

 169      BODY            UNICL

 170      RTNSYM          'EXPR'
                          'TWOPOINTS'

 171      CLSTARCL        CLOCL
                          COLCL
                          CONDCL
                          CASECL

 172      CLOCL           'LEFTPAR' SERCL 'RIGHTPAR'
                          'BEGIN' SERCL 'END'
                          'ELEM' OPEN SERCL CLOSE

 173      COLCL           'LEFTPAR' CLARRAY 'RIGHTPAR'
                          'BEGIN' CLARRAY 'END'
                          'PAR' OPEN CLARRAY CLOSE

 174      CLARRAY         UNICL 'COMMA' UNICL
                          CLARRAY 'COMMA' UNICL

 175      CONDCL          'IF' SERCL CHOICL 'FI'
                          'LEFTPAR' SERCL CHOICL 'RIGHTPAR'
```

176	CHOICL	'THEN' SERCL 'ELSE' SERCL 'THEN' SERCL 'ELSESYMBOL' SERCL 'ELSESYMBOL' SERCL 'ELSESYMBOL' SERCL 'THEN' SERCL 'ELSF' SERCL CHOICL 'THEF' SERCL CHOICL 'ELSESYMBOL' SERCL 'ELSEIFSYMBOL' SERCL CHOICL 'ELSEIFSYMBOL' SERCL CHOICL
177	CASECL	'LEFTPAR' UNICL 'ELSESYMBOL' CLARRAY 'RIGHTPAR' 'CASE' UNICL 'IN' CLARRAY 'ESAC'
178	FORCL	FORFRCM BYTO WHDO FORFROM WHDO BYTO WHDO WHDO
179	FORFROM	FORID FROMCL FORID FROMCL
180	BYTO	BYCL TOCL BYCL TOCL
181	WHDO	WHCL DOCL DOCL
182	FORID	'FOR' 'IDEN'
183	FROMCL	'FROM' UNICL
184	BYCL	'BY' UNICL
185	TOCL	'TO' UNICL
186	WHCL	'WHILE' UNICL
187	DOCL	'DO' UNICL

```
RECURSIVE CLASSNAME

            75

CALL LOOPS L =   1

( 75 ( 76 188 ))
( 188 ()( 76 188 ))

RECURSIVE CLASSNAME

            78

CALL LOOPS L =   1

( 78 ( 83 8 189 )( 137 8 189 ))
( 189 ()( 83 8 189 )( 137 8 189 ))

RECURSIVE CLASSNAME

            79

CALL LOOPS L =   1


COLLECT GARBAGE
( 79 ( 78 81 8 190 )( 78 80 82 190 )( 81 8 190 )( 80 82 190 ))
( 190 ()( 80 8 190 )( 80 82 190 ))

RECURSIVE CLASSNAME

            83

CALL LOOPS L =   1

( 83 ( 84 191 ))
( 191 ()( 10 84 191 ))

RECURSIVE CLASSNAME

            88
```

```
CALL LOOPS L =    1

( 88 ( 85 10 12 13 121 192 ))
( 192 ()( 10 12 13 121 192 ))

RECURSIVE CLASSNAME

             89

CALL LOOPS L =    1

( 89 ( 86 10 12 13 109 193 ))
( 193 ()( 10 12 13 109 193 ))

RECURSIVE CLASSNAME

             90

CALL LOOPS L =    1

( 90 ( 87 10 12 13 115 194 ))
( 194 ()( 10 12 13 115 194 ))

RECURSIVE CLASSNAME

             92

CALL LOOPS L =    1


COLLECT GARBAGE
( 92 ( 91 10 17 13 18 195 ))
( 195 ()( 10 17 13 18 195 ))

RECURSIVE CLASSNAME

             99

CALL LOOPS L =    1

( 99 ( 94 10 96 196 ))
( 196 ()( 10 96 196 ))

RECURSIVE CLASSNAME

            100

CALL LOOPS L =    1


COLLECT GARBAGE
( 100 ( 95 10 97 197 ))
( 197 ()( 10 97 197 ))

RECURSIVE CLASSNAME

            101
```

```
CALL LOOPS L =    1

( 101 ( 135 96 198 ))
( 198 ()( 10 105 198 ))

RECURSIVE CLASSNAME

            102

CALL LOOPS L =    1

( 102 ( 135 97 199 ))
( 199 ()( 10 106 199 ))

RECURSIVE CLASSNAME

            103

CALL LOOPS L =    1


COLLECT GARBAGE
( 103 ( 136 96 200 ))
( 200 ()( 10 107 200 ))

RECURSIVE CLASSNAME

            104

CALL LOOPS L =    1

( 104 ( 136 97 201 ))
( 201 ()( 10 108 201 ))

RECURSIVE CLASSNAME

            110

CALL LOOPS L =    1


COLLECT GARBAGE
( 110 ( 111 202 ))
( 202 ()( 10 111 202 ))

RECURSIVE CLASSNAME

            112

CALL LOOPS L =    1

( 112 ( 121 6 203 ))
( 203 ()( 10 6 203 ))

RECURSIVE CLASSNAME

            113
```

```
CALL LOOPS L =    1

COLLECT GARBAGE
( 113 ( 14 109 6 10 109 6 204 ))
( 204 ()( 10 109 6 204 ))

RECURSIVE CLASSNAME

            114

CALL LOOPS L =    1

( 114 ( 15 115 6 10 115 6 205 ))
( 205 ()( 10 115 6 205 ))

RECURSIVE CLASSNAME

            116

CALL LOOPS L =    1

COLLECT GARBAGE
( 116 ( 121 206 ))
( 206 ()( 10 121 206 ))

RECURSIVE CLASSNAME

            117

CALL LOOPS L =    1

( 117 ( 20 119 207 ))
( 207 ()( 10 119 207 ))

RECURSIVE CLASSNAME

            118

CALL LOOPS L =    1

COLLECT GARBAGE
( 118 ( 117 10 119 208 ))
( 208 ()( 10 119 208 ))

RECURSIVE CLASSNAME

            126

CALL LOOPS L =    1

( 126 ( 28 209 ))
( 209 ()( 28 209 ))

RECURSIVE CLASSNAME
```

Recherche de certaines récursivités cachées

Cette recherche a été introduite dans le programme REC. Nous pouvons trouver des symboles non-terminaux récursifs à gauche de manière cachée :

- on calcule la fermeture transitive de la relation de dépendance à gauche ℓ
- on calcule la fermeture transitive de la relation de dépendance à gauche apparente ℓ_a. Soient PREMIER (i, j) et PREMIERA (i, j) respectivement les éléments des matrices booléennes associées à ℓ^+ et ℓ_a^+.

Si PREMIER (i, i) = vrai et PREMIERA (i, i) = faux alors le symbole X_i est récursif à gauche de manière cachée.

On ne trouve pas ainsi toutes les récursivités à gauche cachées car, une telle récursivité peut être à la fois cachée et apparente.

Exemple

$X_1 \rightarrow X_2X_1 \mid X_1a \mid b$

$X_2 \rightarrow \varepsilon$

Si, après l'exécution du programme REC, il reste des récursivités cachées, elles sont signalées par les programmes SF ou TGA.

6.2. Programme ULD. Application à la syntaxe de PL/1

6.2.1.

La syntaxe de PL/1, ULD version III [Urs] a été traduite en Backus, puis soumise au test des conditions LL(1). (programmes ULD / TGA).

On a pris comme grammaire d'entrée l'ensemble des règles de haut niveau ('higher level productions'). La liste des symboles terminaux est formée de la liste des mots de PL/1 ('PL/1 words'), à laquelle nous avons du ajouter quatre symboles :

"letter" à cause de la règle (19)

"external-option", "env-option", "incorporate-specification" qui ne sont pas définis dans la syntaxe.

Il y a en tout 217 symboles terminaux.

La grammaire d'entrée a 147 règles.

La grammaire BNF obtenue a 316 règles.

Ce dernier nombre pourrait être diminué en réduisant les règles qui ont des parties droites identiques, mais ceci est facile à faire à la main en s'aidant de la table de références croisées.

Les tests des conditions LL(1) ont permis d'indiquer les productions non LL(1), mais aussi quelques erreurs (absence de ";" à la fin de certaines instructions).

La non-considération des règles de bas niveau ('lower level productions') entraîne que "identifier" est considéré comme un terminal. Dans la grammaire, la distinction est donc faite entre symbole terminal et identificateur (ce qui n'est pas vrai au niveau du langage PL/1). Ce dernier problème a déjà été étudié [Gri4 p.68] [Ter].

6.2.2. Résultats

Les symboles récursifs à gauche sont :

expression	règle n°	133
expression-six		134
expression-five		135
expression-four		137
expression-three		138
expression-two		139
reference		142

Les règles suivantes ne satisfont pas aux conditions LL(1) 2 ou 3 :

sentencelist	règle n°	4
sentence		7
default-sentence		13
bound-pair		28
initial-item		45
initial-iteration		46
arithmetic-init-constant		48
complex-format		67
statement		72
labellist		77

group	règle n°	81
if-statement		86
balanced-statement		88
allocate-statement		101
enable-statement		116
datalist-element		129
expression-one		140
primitive-expression		141
basic-reference		143

Toutes les règles satisfont à la condition LL(1) 4.

6.2.3. Etude des règles qui ne sont pas LL(1)

1°) Les récursivités à gauche sont éliminées suivant l'algorithme du paragraphe 2.2.3. Remarquons qu'elles sont toutes des récursivités d'ordre 1 (cliques maximales ayant un seul élément).

2°) La règle 116 n'est pas LL(1) à cause du symbole ACCESS. Cela est du à l'omission de ";" à la fin de l'instruction "enable-statement".

Ce symbole ";" est aussi omis dans les règles

incorporate-statement	93
fetch-statement	94
release-statement	95
disable-statement	117

3°) Les règles 13, 28, 48, 67, 81, 101, 116, 140, 143 peuvent être facilement transformées en LL(1).

4°) Les autres règles demandent un examen plus approndi.
On peut écrire en LL(1) les règles n° 4, 7, 72, 77, 88. La principale difficulté est due à l'élément :

(labellist) assignment-statement

en effet labellist dérive sur basic-reference :
et assignment-statement dérive sur basic-reference =
basic-reference ,
basic-reference - >

Il faut alors substituer assignment-statement et labellist, puis factoriser basic-reference.

Les règles 45, 46, 141 peuvent être réécrites en LL(1) au prix d'un long travail. La difficulté est de discriminer les éléments du langage :

(expression)

et

(integer) simple-string-constant

en effet expression peut dériver sur "real-constant" puis "integer".
La transformation se fait par substitutions-factorisations.
Sur une règle simplifiée :

expression → (expression) |
(integer) simple-string-constant |
integer

on obtient :

expression → (expression-continuation | integer
expression-continuation → (expression-continuation) |
integer) (simple-string-constant)

qui sont des règles LL(1).

5°) La règle 129

datalist-element →

(datalist DO do-specification) | expression

génère un langage qui est non LL(k) quelque soit k, mais cette règle est LR(1). Ce cas a déjà été signalé [Gri4 p. 42].

Nous avons trouvé que la règle 86, que l'on peut écrire :

if-statement →

if-clause < statement | balanced-statement ELSE statement >

génère aussi un langage non LL(k) quel que soit k.

Il n'est pas difficile de montrer que cette règle est LR(1) en utilisant une condition nécessaire et suffisante de Knuth [Kn1].

Nous allons démontrer que le langage n'est pas LL(k) pour tout $k \geq 1$.

La syntaxe de l'instruction de PL/1 peut être réécrite (on fait abstraction de "labellist" ou "prefixlist" qui ne jouent pas de rôle dans ce qui suit).

statement → if-clause statement |
if-clause balanced-statement ELSE statement |
unconditional-statement

balanced-statement → if-clause balanced-statement ELSE
balanced-statement |
unconditional-statement
if-clause → IF expression THEN

unconditional-statement ne peut pas commencer par "IF expression THEN" ou "ELSE", on peut donc se ramener à l'étude de la grammaire :

S → aS | aTbS | c
T → aTbT | c

où :

S représente statement
R " balanced-statement
a " if-clause
b " ELSE
c " unconditional-statement

Toute chaîne générée par cette grammaire a la forme :

$x_{n,\,o} = a^n c \qquad n \geq 0$

ou

$x_{n,\,m} = a^n c b x_1 b x_2 b \ldots x_i b \ldots x_m b y \qquad m < n$

x_i est une chaîne qui commence par "a" et qui finit par "c" et qui possède autant de "a" que de "b", et qui ne dépend ni de n, ni de m
y est une chaîne qui est de la forme $a^p c$ ou bien qui possède un nombre de "a" supérieur ou égal au nombre des "b" et y ne dépend ni de n, ni de m.

Le nombre des "b" représentés dans la chaîne $x_{n,\,m}$ soit m + 1 doit être inférieur ou égal au nombre des "a" représentés dans cette même chaîne, soit n. Nous sommes ramenés à une forme de chaîne tout à fait analogue au langage $\{a^n b^m : 0 \leq m \leq n\}$. Ce dernier langage est non LL(k) quel que soit k [Ros2].

6.2.4. Remarques

L'étude de la syntaxe de PL/1 sous forme ULD nous a conduit à faire les remarques suivantes :

1°) Les symboles "real-constant", "imaginary-constant", "sterling-constant" sont considérés comme symboles terminaux dans les règles de haut-niveau. Dans les règles de bas-niveau ils dérivent sur le symbole "integer" lequel est présent dans la grammaire de haut-niveau et joue un rôle dans les conditions LL(1) (règles n° 45, 46, 141). Il serait donc logique de placer les 'règles de bas-niveau' n° 154, 155, 156, 157, 163 dans la grammaire de haut-niveau.

2°) Le langage généré par la grammaire étudiée n'est pas LL(1) à cause de "if-statement" et de "datalist-element".

3°) La grammaire est certainement non ambiguë, car les seules règles qu'il n'est pas possible de mettre sous forme LL(1) sont des règles LR(1), donc non ambiguës.

Mais nous n'avons pas testé si la grammaire toute entière était LR(1).

```
INPUT SYNTAX=
1   PROGRAM         PROCEDURECL +
2   PROCEDURECL     ( PREFIXLIST ) LABELLIST 'PROCEDURE' ( PARAMETERLIS ) ( PROCEDUREOPT ) 'SEMICOLON'
                     SENTENCELIST
3   PARAMETERLIS    'LEFTBRACKET' < . 'COMMA' . 'IDENTIFIER' . > 'RIGHTBRACKET'
4   PROCEDUREOPT    < OPTIONSATTRI , RETURNSATTRI , 'ORDER' , 'REORDER' , 'RECURSIVE' > +
5   SENTENCELIST    ( SENTENCE + ) ENDCLAUSE
6   ENDCLAUSE       ( PREFIXLIST ) ( LABELLIST ) 'END' 'SEMICOLON'
7   SENTENCE        PROCEDURECL , ENTRYCL , DECLARATIONS , FORMATSENTEN , STATEMENT
8   ENTRYCL         LABELLIST 'ENTRY' ( PARAMETERLIS ) ( RETURNSATTRI ) 'SEMICOLON'
                    ***************************************************************************
                    *
                    *                        1 DECLARATIONS
                    *
                    ***************************************************************************

9   DECLARATIONS    ( LABELLIST ) < DECLARESENTE , DEFAULTSENTE >
10  DECLARESENTE    'DECLARE' DECLARATIONL 'SEMICOLON'
11  DECLARATIONL    < . 'COMMA' . DECLARATION . >
12  DECLARATION     ( 'INTEGER' ) < 'IDENTIFIER' , 'LEFTBRACKET' DECLARATIONL 'RIGHTBRACKET' >
                     ( DIMENSIONATT ) ( ATTRIBUTE + )
13  DEFAULTSENTE    DEFAULTOPT1 , DEFAULTOPT2
14  DEFAULTOPT1     'DEFAULT' 'ALL' ( ATTRIBUTESPE ) 'SEMICOLON'
15  DEFAULTOPT2     'DEFAULT' < . 'COMMA' . DEFAULTSPEC . > 'SEMICOLON'
16  DEFAULTSPEC     SIMPLEDEFAUL , FACTOREDDEFA
17  SIMPLEDEFAUL    RANGESPEC ( ATTRIBUTESPE )
18  RANGESPEC       IDENTIFIERRA , 'DESCRIPTORS'
19  IDENTIFIERRA    'RANGE' 'LEFTBRACKET'
                     < < . 'COMMA' . < 'IDENTIFIER' , 'LETTER' 'COLON' 'LETTER' > . > , 'STAR' >
                     'RIGHTBRACKET'
20  ATTRIBUTESPE    ( DIMENSIONATT ) < ATTRIBUTE , VALUECLAUSE > + , 'SYSTEM'
21  VALUECLAUSE     'VALUE' 'LEFTBRACKET' < . 'COMMA' . VALUESPEC . > 'RIGHTBRACKET'
22  FACTOREDDEFA    'LEFTBRACKET' < . 'COMMA' . DEFAULTSPEC . > 'RIGHTBRACKET' ( ATTRIBUTESPE )
```

```
3   VALUESPEC      PRECISIONSPE , STRINGATTRIB , AREAATTRIBUT
4   PRECISIONSPE   ARITHMETICAT + , 'LEFTBRACKET' < . 'COMMA' . ARITHMETICAT . > 'RIGHTBRACKET' ARITHMETICAT
                   ******************************************************************************************
                   *                                                                                        *
                   *                          1.1 ATTRIBUTS                                                 *
                   *                                                                                        *
                   ******************************************************************************************

5   OPTIONSATTRI   'OPTIONS' 'LEFTBRACKET' < . 'COMMA' . 'EXTERNALOPTI' . > 'RIGHTBRACKET'
6   RETURNSATTRI   'RETURNS' 'LEFTBRACKET' < DATAATTRIBUT , ENTRYNAMEATT , 'FILE' > + 'RIGHTBRACKET'
7   DIMENSIONATT   'LEFTBRACKET' < . 'COMMA' . BOUNDPAIR . > 'RIGHTBRACKET'
8   BOUNDPAIR      ( REFEREXPRESS 'COLON' ) REFEREXPRESS , 'STAR'
9   REFEREXPRESS   EXPRESSION ( 'REFER' 'LEFTBRACKET' UNSUBSCRIPTE 'RIGHTBRACKET' )
0   ATTRIBUTE      DATAATTRIBUT , NONDATAATTRI , ENTRYNAMEATT , FILENAMEATTR , SCOPEATTRIBU , LIKEATTRIBUT
1   DATAATTRIBUT   ARITHMETICAT , STRINGATTRIB , 'VARYING' , PICTUREATTRI , AREAATTRIBUT , LABELATTRIBU ,
                    'POINTER' , OFFSETATTRIB , 'TASK' , 'EVENT' , STORAGECLASS , DEFINEDATTRI ,
                    BASEDATTRIBU , 'UNALIGNED' , 'ALIGNED' , 'SECONDARY' , 'CONNECTED' , 'VARIABLE' ,
                    INITIALATTRI
2   ARITHMETICAT   < 'REAL' , 'COMPLEX' , 'DECIMAL' , 'BINARY' , 'FLOAT' , 'FIXED' >
                   ( 'LEFTBRACKET' 'INTEGER' ( 'COMMA' SIGNEDINTEGE ) 'RIGHTBRACKET' )
3   SIGNEDINTEGE   ( 'PLUS' , 'MINUS' ) 'INTEGER'
4   STRINGATTRIB   < 'BIT' , 'CHARACTER' > ( 'LEFTBRACKET' < REFEREXPRESS , 'STAR' > 'RIGHTBRACKET' )
5   PICTUREATTRI   'PICTURE' ( 'PICTURESPECI' )
6   AREAATTRIBUT   'AREA' ( 'LEFTBRACKET' < REFEREXPRESS , 'STAR' > 'RIGHTBRACKET' )
7   LABELATTRIBU   'LABEL' ( 'LEFTBRACKET' < . 'COMMA' . 'IDENTIFIER' . > 'RIGHTBRACKET' )
8   OFFSETATTRIB   'OFFSET' ( 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' )
9   STORAGECLASS   'AUTOMATIC' , 'STATIC' , 'CONTROLLED'
0   DEFINEDATTRI   'DEFINED' BASICREFEREN , 'POSITION' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET'
1   BASEDATTRIBU   'BASED' ( 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' )
2   INITIALATTRI   'INITIAL' < INITIALCALL , INITIALITEML >
3   INITIALCALL    'CALL' REFERENCE
```

```
14  INITIALITEML   'LEFTBRACKET' < . 'COMMA' . INITIALITEM . > 'RIGHTBRACKET'

15  INITIALITEM    INITIALITERA , INITIALCONST , 'SIMPLESTRING' , REFERENCE ,
                    'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' , 'STAR'

16  INITIALITERA   'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' < INITIALCONST , INITIALITEML , REFERENCE >

17  INITIALCONST   REPLICATEDST , ARITHMETICIN , 'STERLINGCONS'

18  ARITHMETICIN   ( 'PLUS' , 'MINUS' ) 'REALCONSTANT' ( < 'PLUS' , 'MINUS' > 'IMAGINARYCON' ) ,

                   ( 'PLUS' , 'MINUS' ) 'IMAGINARYCON'

19  NONDATAATTRI   'BUILTIN' , GENERICATTRI , ATTENTIONATT

20  ENTRYNAMEATT   'ENTRY' ( 'LEFTBRACKET' DESCRIPTORLI 'RIGHTBRACKET' ) ,
                    RETURNSATTRI , 'REDUCIBLE' , 'IRREDUCIBLE'

21  DESCRIPTORLI   DESCRIPTOR ( 'COMMA' DESCRIPTORLI )

22  DESCRIPTOR     ( 'INTEGER' ) ( DIMENSIONATT ) ( ATTRIBUTE + ) , 'STAR'

23  FILENAMEATTR   'FILE' , FILEATTRIBUT , 'ENVIRONMENT' 'LEFTBRACKET' 'ENVOPTION' 'RIGHTBRACKET'

24  FILEATTRIBUT   'BITSTREAM' , 'STREAM' , 'RECORD' , 'INPUT' , 'OUTPUT' , 'UPDATE' ,

                    'SEQUENTIAL' , 'DIRECT' , 'BUFFERED' , 'UNBUFFERED' , 'KEYED' , 'PRINT' ,

                    'BACKWARDS' , 'EXCLUSIVE' , 'TRANSIENT'

25  GENERICATTRI   'GENERIC' 'LEFTBRACKET' < . 'COMMA' . GENERICELEME . > 'RIGHTBRACKET'

26  GENERICELEME   REFERENCE 'WHEN' 'LEFTBRACKET' DESCRIPTORLI 'RIGHTBRACKET'

27  SCOPEATTRIBU   'INTERNAL' , 'EXTERNAL'

28  LIKEATTRIBUT   'LIKE' UNSUBSCRIPTE

29  ATTENTIONATT   'ATTENTION' 'ENVIRONMENT' 'LEFTBRACKET' 'ENVOPTION' 'RIGHTBRACKET'

                   ******************************************************************************
                   *
                   *                        1.2 FORMATS
                   *
                   ******************************************************************************


30  FORMATSENTEN   ( PREFIXLIST ) LABELLIST 'FORMAT' FORMATLIST 'SEMICOLON'

31  FORMATLIST     'LEFTBRACKET' < . 'COMMA' . FORMATCL . > 'RIGHTBRACKET'

32  FORMATCL       FORMATITERAT , FORMATITEM

33  FORMATITERAT   < 'INTEGER' , 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' > < FORMATITEM , FORMATLIST >

34  FORMATITEM     DATAFORMAT , CONTROLFORMA , REMOTEFORMAT
```

```
TAFORMAT      REALFORMAT , COMPLEXFORMA , STRINGFORMAT , PICTUREFORMA

REALFORMAT    < 'E' , 'F' > 'LEFTBRACKET' EXPRESSION
               { 'COMMA' EXPRESSION { 'COMMA' EXPRESSION } } 'RIGHTBRACKET'

COMPLEXFORMA  'C' 'LEFTBRACKET' < REALFORMAT , PICTUREFORMA >
               { 'COMMA' REALFORMAT , 'COMMA' PICTUREFORMA } 'RIGHTBRACKET'

STRINGFORMAT  < 'A' , 'B' , 'BB' > { 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' }

PICTUREFORMA  < 'BP' , 'P' > 'PICTURESPECI'

CONTROLFORMA  < 'BCOLUMN' , 'BX' , 'COLUMN' ,
               'LINE' , 'PAGE' , 'SKIP' , 'X' >
               { 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' }

REMOTEFORMAT  'R' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET'
```

2 INSTRUCTIONS

```
STATEMENT     { PREFIXLIST } { LABELLIST } < IFSTATEMENT , UNCONDITIONA >

PREFIXLIST    < 'LEFTBRACKET' < . 'COMMA' . PREFIXELEMEN . > 'RIGHTBRACKET' 'COLON' > +

PREFIXELEMEN  PREFIX , NOPREFIX , CHECKCONDITI , NOCHECKCONDI

PREFIX        'CONVERSION' , 'FIXEDOVERFLO' , 'OVERFLOW' , 'STRINGRANGE' , 'STRINGSIZE' , 'SUBSCRIPTRAN' ,
               'UNDERFLOW' , 'ZERODIVIDE'

NOPREFIX      'NOCONVERSION' , 'NOFIXEDOVERF' , 'NOOVERFLOW' , 'NOSIZE' , 'NOSTRINGSIZE' , 'NOSTRINGRANG' ,
               'NOSUBSCRIPTR' , 'NOUNDERFLOW' , 'NOZERODIVIDE'

LABELLIST     < BASICREFEREN 'COLON' > +

UNCONDITIONA  BEGINBLOCK , GROUP , GOTOSTATEMEN , CALLSTATEMEN , INCORPORATES , FETCHSTATEME ,
               RELEASESTATE , RETURNSTATEM , WAITSTATEMEN , DELAYSTATEME , EXITSTATEMEN , STOPSTATEMEN ,
               ASSIGNMENTST , ALLOCATESTAT , FREESTATEMEN , ONSTATEMENT , REVERTSTATEM , SIGNALSTATEM ,
               ENABLESTATEM , DISABLESTATE , ACCESSSTATEM , OPENSTATEMEN , CLOSESTATEME , STREAMIOSTAT ,
               RECORDIOSTAT , DISPLAYSTATE , NULLSTATEMEN

NULLSTATEMEN  'SEMICOLON'
```

```
                    ******************************************************************************
                    *
                    *                    2.1 BLOCS ET GROUPES
                    *
                    ******************************************************************************

80  BEGINBLOCK      'BEGIN' ( < OPTIONSATTRI , 'ORDER' , 'REORDER' > + ) 'SEMICOLON' SENTENCELIST

81  GROUP           SIMPLEGROUP , ITERATEDGROU

82  SIMPLEGROUP     'DO' 'SEMICOLON' SENTENCELIST

83  ITERATEDGROU    'DO' < DOSPECIFICAT , 'WHILE' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' >

                     'SEMICOLON' SENTENCELIST

84  DOSPECIFICAT    REFERENCE 'EQUAL' < . 'COMMA' . SPECIFICATIO . >

85  SPECIFICATIO    EXPRESSION ( 'BY' EXPRESSION ( 'TO' EXPRESSION ) ,
                     'TO' EXPRESSION ( 'BY' EXPRESSION ) )
                     ( 'WHILE' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' )

                    ******************************************************************************
                    *
                    *                    2.2 INSTRUCTIONS DE CONTROLE
                    *
                    ******************************************************************************

86  IFSTATEMENT     IFCLAUSE STATEMENT , IFCLAUSE BALANCEDSTAT 'ELSE' STATEMENT

87  IFCLAUSE        'IF' EXPRESSION 'THEN'

88  BALANCEDSTAT    ( PREFIXLIST ) ( LABELLIST ) < IFCLAUSE BALANCEDSTAT 'ELSE' BALANCEDSTAT ,

                     UNCONDITIONA >

89  GOTOSTATEMEN    < 'GOTO' , 'GO' 'TO' > REFERENCE 'SEMICOLON'

90  CALLSTATEMEN    'CALL' REFERENCE ( CALLOPTIONSL ) 'SEMICOLON'

91  CALLOPTIONSL    < 'TASK' ( 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) ,
                     'PRIORITY' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ,
                     'EVENT' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' > +

92  RETURNSTATEM    'RETURN' ( 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ) 'SEMICOLON'

93  INCORPORATES    'INCORPORATE' 'LEFTBRACKET' 'INCORPORSPEC' 'RIGHTBRACKET'

94  FETCHSTATEME    'FETCH' < . 'COMMA' . REFERENCE . >

95  RELEASESTATE    'RELEASE' < . 'COMMA' . REFERENCE . >

96  WAITSTATEMEN    'WAIT' 'LEFTBRACKET' < . 'COMMA' . REFERENCE . > 'RIGHTBRACKET' ( 'LEFTBRACKET'

                     EXPRESSION 'RIGHTBRACKET' ) 'SEMICOLON'

97  DELAYSTATEME    'DELAY' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' 'SEMICOLON'
```

```
98   EXITSTATEMEN    'EXIT' 'SEMICOLON'
99   STOPSTATEMEN    'STOP' 'SEMICOLON'
                     ***********************************************************************
                     *
                     *              2.3 INSTRUCTIONS DE GESTION DE MEMOIRE CENTRALE
                     *
                     ***********************************************************************

100  ASSIGNMENTST    < . 'COMMA' . REFERENCE . > 'EQUAL' EXPRESSION
                      ( 'COMMA' 'BY' 'NAME' ) 'SEMICOLON'
101  ALLOCATESTAT    'ALLOCATE' < . 'COMMA' . < BASEDALLOCAT , CONTROLLEDAL > . > 'SEMICOLON'
102  BASEDALLOCAT    'IDENTIFIER' < 'SET' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET'
                      ( 'IN' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) ,
                      'IN' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET'
                      ( 'SET' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) >
103  CONTROLLEDAL    ( 'INTEGER' ) 'IDENTIFIER' ( DIMENSIONATT )
                      ( < STRINGATTRIB , AREAATTRIBUT , INITIALATTRI > + )
104  FREESTATEMEN    'FREE' < . 'COMMA' . < REFERENCE
                      ( 'IN' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) > . > 'SEMICOLON'
                     ***********************************************************************
                     *
                     *        2.4 INSTRUCTIONS DE TAITEMENT DES 'CONDITION' ET DES 'ATTENTION'
                     *
                     ***********************************************************************

105  ONSTATEMENT     'ON' CONDITIONCL ( 'SNAP' ) < ONCONDITIONA , 'SYSTEM' 'SEMICOLON' >
106  REVERTSTATEM    'REVERT' CONDITIONCL 'SEMICOLON'
107  SIGNALSTATEM    'SIGNAL' CONDITIONCL 'SEMICOLON'
108  CONDITIONCL     PREFIX , CHECKCONDITI , 'AREA' , NAMEDIOCONDI , 'ERROR' , 'FINISH' ,
                      PROGRAMMERNA , ATTENTIONCON
109  CHECKCONDITI    'CHECK' 'LEFTBRACKET' < . 'COMMA' . UNSUBSCRIPTE . > 'RIGHTBRACKET'
110  NOCHECKCONDI    'NOCHECK' 'LEFTBRACKET' < . 'COMMA' . UNSUBSCRIPTE . > 'RIGHTBRACKET'
111  NAMEDIOCONDI    IOCONDITION 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET'
112  IOCONDITION     'BEGINVOLUME' , 'ENDFILE' , 'ENDPAGE' , 'ENDVOLUME' , 'KEY' , 'NAME' ,
                      'PENDING' , 'RECORD' , 'TRANSMIT' , 'UNDEFINEDFIL'
113  PROGRAMMERNA    'CONDITION' 'LEFTBRACKET' 'IDENTIFIER' 'RIGHTBRACKET'
```

```
114  ATTENTIONCON    'ATTENTION' 'LEFTBRACKET' < . 'COMMA' . 'IDENTIFIER' . > 'RIGHTBRACKET'

115  ACCESSSTATEM    'ACCESS' 'ATTENTION' ( 'LEFTBRACKET' < . 'COMMA' . 'IDENTIFIER' . > 'RIGHTBRACKET' )
                     < 'ELSE' STATEMENT , 'SEMICOLON' >

116  ENABLESTATEM    'ENABLE' < . 'COMMA' . < ATTENTIONCON
                      ( 'ACCESS' , 'ASYNC' , 'EVENT' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) + > . >

117  DISABLESTATE    'DISABLE' ATTENTIONCON

                     ******************************************************************************
                     *
                     *                        2.5 INSTRUCTIONS D'ENTREES ET DE SORTIES
                     *
                     ******************************************************************************


118  OPENSTATEMEN    'OPEN' < . 'COMMA' . OPENOPTIONSL . > 'SEMICOLON'

119  OPENOPTIONSL    < FILEATTRIBUT , 'FILE' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' , 'BLINESIZE'
                      'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' , 'LINESIZE' 'LEFTBRACKET' EXPRESSION
                      'RIGHTBRACKET' , 'PAGESIZE' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' , 'TITLE'
                      'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' , 'ENVIRONMENT' 'LEFTBRACKET' 'ENVOPTION'
                      'RIGHTBRACKET' , 'VOLUME' > +

120  CLOSESTATEME    'CLOSE' < . 'COMMA' . CLOSEOPTIONS . > 'SEMICOLON'

121  CLOSEOPTIONS    < 'FILE' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                      'ENVIRONMENT' 'LEFTBRACKET' 'ENVOPTION' 'RIGHTBRACKET' , 'VOLUME' > +

122  STREAMIOSTAT    < 'GET' , 'PUT' > STREAMOPTION 'SEMICOLON'

123  STREAMOPTION    < 'FILE' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                      'BITSTRING' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ,
                      'STRING' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' , DATASPECIFIC , 'COPY' ,
                      'SKIP' ( 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ) , 'PAGE' ,
                      'LINE' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' > +

124  DATASPECIFIC    DATADIRECTED , EDITDIRECTED , LISTDIRECTED

125  DATADIRECTED    'DATA' ( 'LEFTBRACKET' DATALIST 'RIGHTBRACKET' )

126  EDITDIRECTED    'EDIT' < 'LEFTBRACKET' DATALIST 'RIGHTBRACKET' FORMATLIST > +

127  LISTDIRECTED    'LIST' 'LEFTBRACKET' DATALIST 'RIGHTBRACKET'

128  DATALIST        < . 'COMMA' . DATALISTELEM . >

129  DATALISTELEM    'LEFTBRACKET' DATALIST 'DO' DOSPECIFICAT 'RIGHTBRACKET' , EXPRESSION

130  RECORDIOSTAT    < 'READ' , 'WRITE' , 'REWRITE' , 'LOCATE' 'IDENTIFIER' , 'DELETE' ,
```

```
                 'UNLOCK' > RECORDOPTION 'SEMICOLON'

131 RECORDOPTION  < 'FILE' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                  'EVENT' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                  'FROM' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                  'IGNORE' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ,
                  'INTO' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                  'KEY' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ,
                  'KEYTO' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ,
                  'KEYFROM' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' ,
                  'SET' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' , 'NOLOCK' > +

132 DISPLAYSTATE  'DISPLAY' 'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET'
                  ( 'REPLY' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET'
                  ( 'EVENT' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) ,
                  'EVENT' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET'
                  'REPLY' 'LEFTBRACKET' REFERENCE 'RIGHTBRACKET' ) 'SEMICOLON'

                  ************************************************************************************
                  *
                  *                          3 EXPRESSIONS
                  *
                  ************************************************************************************


133 EXPRESSION    EXPRESSIONSI , EXPRESSION 'OR' EXPRESSIONSI

134 EXPRESSIONSI  EXPRESSIONFI , EXPRESSIONSI 'AND' EXPRESSIONFI

135 EXPRESSIONFI  EXPRESSIONFO , EXPRESSIONFI COMPARISONOP EXPRESSIONFO

136 COMPARISONOP  'GT' , 'GE' , 'EQUAL' , 'LT' , 'LE' , 'NG' ,  .
                   'NE' , 'NL'

137 EXPRESSIONFO  EXPRESSIONTH , EXPRESSIONFO 'CAT' EXPRESSIONTH

138 EXPRESSIONTH  EXPRESSIONTW , EXPRESSIONTH < 'PLUS' , 'MINUS' > EXPRESSIONTW

139 EXPRESSIONTW  EXPRESSIONON , EXPRESSIONTW < 'STAR' , 'SLASH' > EXPRESSIONON

140 EXPRESSIONON  PRIMITIVEEXP , < 'PLUS' , 'MINUS' , 'NOT' > EXPRESSIONON ,
                   PRIMITIVEEXP 'EXP' EXPRESSIONON

141 PRIMITIVEEXP  'LEFTBRACKET' EXPRESSION 'RIGHTBRACKET' , REFERENCE , CONSTANT , 'ISUB'

142 REFERENCE     ( REFERENCE 'PT' ) BASICREFEREN

143 BASICREFEREN  ( < 'IDENTIFIER' ( SUBSCRIPTLIS ) 'POINT' > + ) 'IDENTIFIER' ( SUBSCRIPTLIS + )

144 SUBSCRIPTLIS  'LEFTBRACKET' < . 'COMMA' . < EXPRESSION , 'STAR' > . > 'RIGHTBRACKET'

145 UNSUBSCRIPTE  < . 'POINT' . 'IDENTIFIER' . >

146 CONSTANT      'REALCONSTANT' , 'IMAGINARYCON' , 'STERLINGCONS' , 'SIMPLESTRING' , REPLICATEDST

147 REPLICATEDST  'LEFTBRACKET' 'INTEGER' 'RIGHTBRACKET' 'SIMPLESTRING'
```

BNF SYNTAX =

		366	() PROCEDURECL 366
1	219	PROGRAM	PROCEDURECL 366
		367	PREFIXLIST ()
		368	PARAMETERLIS ()
		369	PROCEDUREOPT ()
2	220	PROCEDURECL	367 LABELLIST 'PROCEDURE' 368 369 'SEMICOLON' SENTENCELIST
		370	() 'COMMA' 'IDENTIFIER' 370
3	223	PARAMETERLIS	'LEFTBRACKET' 'IDENTIFIER' 370 'RIGHTBRACKET'
		371	OPTIONSATTRI RETURNSATTRI 'ORDER' 'REORDER' 'RECURSIVE'
		372	() 371 372
4	224	PROCEDUREOPT	371 372
		373	SENTENCE 373 ()
5	225	SENTENCELIST	373 ENDCLAUSE
		374	PREFIXLIST ()
		375	LABELLIST ()
6	229	ENDCLAUSE	374 375 'END' 'SEMICOLON'
7	228	SENTENCE	PROCEDURECL ENTRYCL DECLARATIONS FORMATSENTEN STATEMENT
		376	PARAMETERLIS ()
		377	RETURNSATTRI

			()
8	230	ENTRYCL	LABELLIST 'ENTRY' 376 377 'SEMICOLON'
		378	LABELLIST ()
		379	DECLARESENTE DEFAULTSENTE
9	231	DECLARATIONS	378 379
10	234	DECLARESENTE	'DECLARE' DECLARATIONL 'SEMICOLON'
		380	() 'COMMA' DECLARATION 380
11	236	DECLARATIONL	DECLARATION 380
		381	'INTEGER' ()
		382	'IDENTIFIER' 'LEFTBRACKET' DECLARATIONL 'RIGHTBRACKET'
		383	DIMENSIONATT ()
		384	ATTRIBUTE 384 ()
12	237	DECLARATION	381 382 383 384
13	235	DEFAULTSENTE	DEFAULTOPT1 DEFAULTOPT2
		385	ATTRIBUTESPE ()
14	240	DEFAULTOPT1	'DEFAULT' 'ALL' 385 'SEMICOLON'
		386	() 'COMMA' DEFAULTSPEC 386
15	241	DEFAULTOPT2	'DEFAULT' DEFAULTSPEC 386 'SEMICOLON'
16	243	DEFAULTSPEC	SIMPLEDEFAUL FACTOREDDEFA
		387	ATTRIBUTESPE ()
17	244	SIMPLEDEFAUL	RANGESPEC 387
18	246	RANGESPEC	IDENTIFIERRA 'DESCRIPTORS'
		390	'IDENTIFIER'

```
                                'LETTER' 'COLON' 'LETTER'

            389                 ().
                                'COMMA' 390 389

            388                 390 389
                                'STAR'

19    247   IDENTIFIERRA        'RANGE' 'LEFTBRACKET' 388 'RIGHTBRACKET'

            391                 DIMENSIONATT
                                ()

            392                 ATTRIBUTE
                                VALUECLAUSE

            393                 ()
                                392 393

20    242   ATTRIBUTESPE        391 392 393
                                'SYSTEM'

            394                 ()
                                'COMMA' VALUESPEC 394

21    248   VALUECLAUSE         'VALUE' 'LEFTBRACKET' VALUESPEC 394 'RIGHTBRACKET'

            395                 ()
                                'COMMA' DEFAULTSPEC 395

            396                 ATTRIBUTESPE
                                ()

22    245   FACTOREDDEFA        'LEFTBRACKET' DEFAULTSPEC 395 'RIGHTBRACKET' 396

23    249   VALUESPEC           PRECISIONSPE
                                STRINGATTRIB
                                AREAATTRIBUT

            397                 ()
                                ARITHMETICAT 397

            398                 ()
                                'COMMA' ARITHMETICAT 398

24    250   PRECISIONSPE        ARITHMETICAT 397
                                'LEFTBRACKET' ARITHMETICAT 398 'RIGHTBRACKET' ARITHMETICAT

            399                 ()
                                'COMMA' 'EXTERNALOPTI' 399

25    226   OPTIONSATTRI        'OPTIONS' 'LEFTBRACKET' 'EXTERNALOPTI' 399 'RIGHTBRACKET'
```

```
            128

CALL LOOPS L =    1


COLLECT GARBAGE
( 128 ( 10 210 ))
( 210 ()( 10 129 210 )( 10 210 ))

RECURSIVE CLASSNAME

            140
            157
            158

CALL LOOPS L =    3


COLLECT GARBAGE
( 140 ( 171 211 )( 166 214 )( 6 214 ))
( 211 ()( 159 161 160 217 )( 159 160 217 ))
( 212 ( 159 161 160 218 )( 159 160 218 ))
( 213 ( 159 161 160 219 )( 159 160 219 ))
( 157 ( 171 212 )( 166 215 )( 6 215 ))
( 214 ( 211 ))
( 215 ()( 212 ))
( 216 ( 213 ))
( 158 ( 171 213 )( 166 216 )( 6 216 ))
( 217 ( 214 ))
( 218 ( 215 ))
( 219 ()( 216 ))

RECURSIVE CLASSNAME

            151

CALL LOOPS L =    1


COLLECT GARBAGE
( 151 ( 152 17 152 220 ))
( 220 ()( 17 152 220 ))

RECURSIVE CLASSNAME

            161

CALL LOOPS L =    1


COLLECT GARBAGE
( 161 ( 162 221 )( 163 221 )( 10 221 ))
( 221 ()( 10 221 )( 10 162 221 )( 10 163 221 ))

RECURSIVE CLASSNAME

            174


CALL LOOPS L =    1


COLLECT GARBAGE
( 174 ( 137 10 137 222 ))
( 222 ()( 10 137 222 ))
```

C O N C L U S I O N

CONCLUSION

Le travail présenté dans cet ouvrage a obéi à deux préoccupations principales. L'une, théorique, était d'étudier les transformations d'une grammaire context-free quelconque en une grammaire LL(1). L'autre, pratique, était de réaliser des programmes facilement utilisables pour qui veut écrire un compilateur pour un langage donné ou même définir un nouveau langage et l'implémenter.

Examinons d'abord le point de vue théorique. Nous avons proposé deux améliorations possibles de la méthode de Foster.

L'une est déduite de la méthode de Paull et Unger (cf. §3.3 et 3.4.2), l'autre est exposée au §3.4.1. Malheureusement elles ne s'appliquent que dans un nombre limité de cas et nous ne les avons pas introduites dans les programmes. Une étude plus approfondie permettrait peut-être d'élargir leur champ d'application. Ceci serait sans doute possible en étudiant le problème plus général de la transformation d'une grammaire C.F. en une grammaire LL(k), $k \geq 1$. On connaît peu de choses sur ce problème.

Citons les résultats de Kurki-Suonio [Kur2] relatifs à la réduction du "look-ahead". Mais, comme dans le cas des langages LL(1), on ne peut pas décider si une grammaire context-free quelconque génère un langage LL(k), même pour un k fixé Ros2 .

Toutefois la réalisation d'un analyseur d'un langage LL(k), $k > 1$ n'a qu'un intérêt théorique. En effet cela semble peu intéressant pour un langage de programmation. Nous n'avons encore jamais trouvé le cas d'un langage de programmation qui serait LL(k) pour un certain $k > 1$ et qui serait non LL(1). L'étude des propriétés des grammaires et langages LL(1), LL(k),$k > 1$, des s-grammaires et s-langages a été faite en parallèle. Il reste un certain nombre de problèmes ouverts pour ces grammaires. Celui envisagé à la fin du paragraphe 3.4.2 en fait partie.

Passons maintenant au point de vue pratique. Notre but serait effectivement atteint si nos programmes étaient utilisés par ceux qui écrivent des compilateurs. Une note technique doit être rédigée pour l'utilisation de l'ensemble des programmes de transformations de grammaire.

Comme il a été dit en § 3.2.7, il existe un nombre encore important de règles non transformées automatiquement en règles LL(1) (principalement les règles ne satisfaisant pas à la condition LL(1) n° 3). Il reste donc du travail à faire à la main avant de mettre au point une grammaire LL(1).

Dans ce cas la marche à suivre est la suivante.
Lorsqu'on n'a pas réussi en utilisant les programmes à mettre une grammaire sous forme LL(1), on doit se poser la question de savoir si le langage généré est LL(1) ou non. Il n'existe pas bien sûr de méthode générale, puisque cette question est indécidable. Cette besogne est cependant facilitée par l'existence, dans les articles concernant les langages LL(k), de nombreux exemples de langages qui sont non LL(k) pour tout k. On peut essayer de trouver une analogie avec ces langages. On peut se référer à la famille de langages définie à la fin de § 3.2.7.

On peut alors se trouver dans l'une des trois situations suivantes :

1°) le langage est LL(1) et nous pouvons trouver des règles équivalentes et LL(1) et on a donc réussi.

2°) le langage est LL(1), mais la mise sous forme LL(1) conduit à un trop grand nombre de règles.

3°) le langage n'est pas LL(1), ou bien nous ne pouvons pas arriver à montrer qu'il l'est ou qu'il ne l'est pas.

Dans les deux derniers cas 2°) et 3°), on emploie un artifice. On cherche un sur-ensemble du langage qui soit LL(1) et l'on vérifiera la conformité syntaxique grâce à des <u>fonctions</u> sémantiques incorporées dans les règles.

A notre connaissance il n'est pas possible de réaliser cette opération automatiquement. On pourrait formuler le problème de la manière suivante : étant donné un langage non LL(1), comment trouver le plus petit sur-ensemble de ce langage qui soit LL(1) ?

A P P E N D I C E A

APPENDICE A

Grammaire LL(1) pour le langage $\{xcx^R : x \in \{a, b\}^*\}$

$S \rightarrow aSa \mid bSb \mid c \qquad V_N = \{S\} \qquad V_T = \{a, b, c\}$

Analyseur sous forme de programme

(ensemble de routines écrites dans le langage de macro-instructions et qui s'appellent récursivement)

```
ROUTINE S
     DECIDE 'a' , ALT1
     DECIDE 'b' , ALT2
     DECIDE 'c' , ALT3
ALT1 : CHECK 'a'
       CALL S
       CHECK 'a'
       EXIT
ALT2 : CHECK 'b'
       CALL S
       CHECK 'b'
       EXIT
ALT3 : CHECK 'c'
RETURN
```

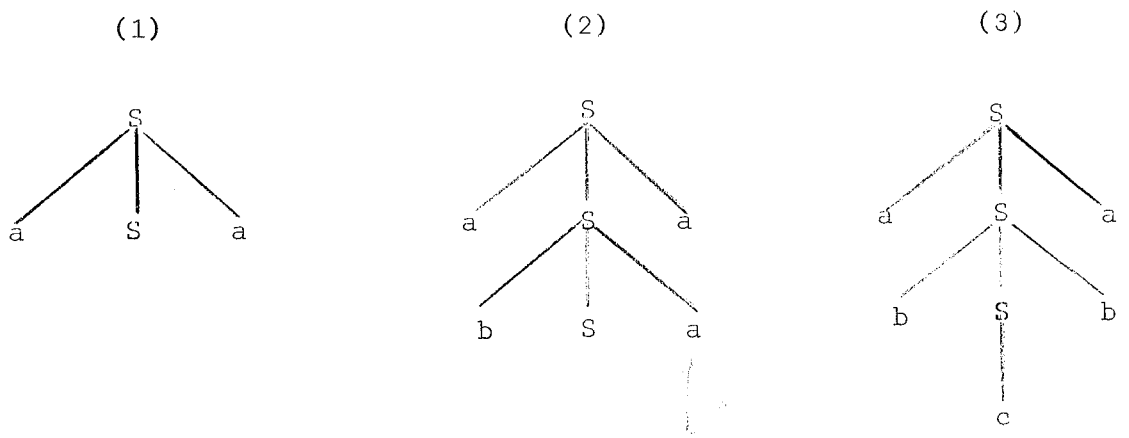
DECIDE provoque un saut à l'étiquette indiquée si le caractère courant est le symbole spécifié.

CHECK teste la présence du symbole indiqué et place le symbole de base suivant comme caractère courant. Si le test est négatif alors il y a erreur.

Analyse d'une chaîne avec comme modèle l'automate à 2 piles [Grif]

chaîne d'entrée	pile d'analyse	actions
abcba ‡	S ‡	(configuration initiale) appel de S, décision
abcba ‡	aSa ‡	test de a
bcba ‡	Sa ‡	appel de S, décision
bcba ‡	bSba ‡	test de b
cba ‡	Sba ‡	appel de S, décision
cba ‡	cba ‡	test de c
ba ‡	ba ‡	test de b
a ‡	a ‡	test de a
‡	‡	(configuration finale)

Dans cette analyse, la structure de la chaîne s'établit de la manière suivante :



A P P E N D I C E B

APPENDICE B

Définition des grammaires LL(k) [Kn2]

Une grammaire hors-contexte est LL(k) si la condition qui suit est vérifiée pour tout x_1, x_4, x_4' dans V_T^* et tout α_2, α_3, α_2', α_3' dans $(V_N \cup V_T)^*$:

$$S \overset{L*}{\rightarrow} x_1 A \alpha_3 \overset{L}{\rightarrow} x_1 \alpha_2 \alpha_3 \overset{L*}{\rightarrow} x_1 x_4$$

et $$S \overset{L*}{\rightarrow} x_1 A \alpha_3' \overset{L}{\rightarrow} x_1 \alpha_2' \alpha_3' \overset{L*}{\rightarrow} x_1 x_4'$$

et $$k : x_4 = k : x_4'$$

implique que

$$\alpha_2 = \alpha_2' .$$

($k : \alpha$ = si $\ell(\alpha) > k$ alors les k premiers caractères de α
sinon α).

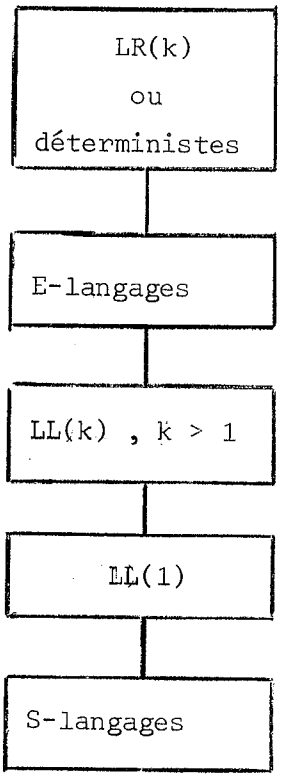
Une règle $A \rightarrow \alpha_2$ satisfaisant à la définition est dite LL(k).

Les références concernant les S-langages et les langages LL(k) sont, par ordre chronologique :
[Kor, Lew, Kn2, Pau, Kur2, Ros2, Wo2, Wo4, Wo5].

APPENDICE C

APPENDICE C

Hiérarchie de langages déterministes



automate à pile déterministe [Kn1, Gin4]

automate à pile déterministe sans ε [Wo2]

machine de Knuth sans retour arrière [Kn2]

s-machine [Kor]

Chaque ensemble de langages est strictement contenu dans celui qui est au-dessus. De plus l'ensemble des langages LL(k) est strictement contenu dans l'ensemble des langages LL(k + 1) pour $k \geq 1$. Remarquons que l'on ne connaît pas d'automate à pile déterministe reconnaissant les langages LL(k) et seulement ceux-ci et que la machine de Knuth la plus générale reconnaît des langages qui ne sont pas context-free.

Si R est un langage régulier alors $R \not\ell$ est un S- langage [Kor].

Propriétés de fermeture [Kor, Ros2, Gin4, Hop]

	S-langages	langages LL(k)	langages LR(k) (déterministes)	langages hors-contexte
union	non	non (1)	non	oui
intersection	non	non	non	non
complément	non	non (2)	oui	non
inverse	non	non	non	oui
intersection avec un ensemble régulier	non	non	oui	oui
union avec un ensemble régulier	non	non	oui	oui
produit	oui	non	non	oui
opération étoile	non	non	non	oui

(1) Si l'union finie de langages LL(k) disjoints forme un ensemble régulier, alors tous les langages sont réguliers [Ros2].

(2) Le complément d'un langage LL(k) non régulier n'est jamais LL(k). [Ros2].

Décidabilité [Kor, Ros2, Gin4, Hop]

Question	S-langages	langages LL(k)	langages LR(k) (déterministes)	langages hors-contexte
L(G) est un ensemble régulier	D	D	D	N
L(G) = R ensemble régulier donné	D	D	D	N
L(G) est vide, fini, infini	D	D	D	D
$L(G) = V_T^*$	D (1)	D	D	N
$L(G_1) = L(G_2)$	D	D (2)	?	N
$L(G_1) \subseteq L(G_2)$	?	?	N	N
$L(G_1) \cap L(G_2)$ est vide	N	?	N	N

D = décidable N = non décidable ? = on ne sait pas

(1) un S-langage ne peut pas être égal à V_T^* (un S-langage ne peut pas contenir la chaîne vide) mais peut être égale à $V_T^* \not\#$.

(2) la démonstration [Ros2] n'est pas convaincante.

Autres questions

Un langage hors-contexte arbitraire est-il un langage LR(k) ? un langage LL(k) ? un S-langage ?

ne sont pas décidables, même pour une valeur fixée de k.
[Gin4, Ros2, Kor].

Existe-t-il une valeur k, $k \geq 1$, telle que une grammaire hors-contexte arbitraire soit LR(k) ? LL(k) ?

ne sont pas décidables
[Kn1, Ros2].

Existe-t-il une valeur k', $k' \geq 1$ telle que G grammaire LR(k) soit LL(k') ?

est décidable
[Ros2].

B I B L I O G R A P H I E

B I B L I O G R A P H I E

{Abr} H.D. ABRAMSON "A note on left recursive rules and the partitioning of a recognition matrix for syntax-directed translation". BIT Vol.10 p. 1-5 1970.

{Ass} M. ASSABGUI "Notation SRL et génération automatique d'analyseurs". Note technique I.M.A.G. mai 1970.

{Bor1} J. BORDIER "Elimination de la récursivité à gauche dans une grammaire context-free". Note technique I.M.A.G. février 1970 & R.I.R.O. B n° 3 - 1970 (à paraître).

{Bor2} J. BORDIER "Notation ULD et analyse descendante déterministe". Note technique C.S. IBM GRENOBLE (à paraître).

{Bor3} J. BORDIER & H. SAYA "Note sur les langages LL(k)". I.M.A.G. 1970 (à paraître).

{Bou} R. BOUDEAUD "Etude et réalisation d'un compilateur ALGOL pour calculateur PB250-A100". Thèse Clermont-Ferrand 1968.

{Bous} J.C. BOUSSARD "Cours d'analyse syntaxique". Maîtrise d'Informatique. Faculté des Sciences. GRENOBLE 1970.

{Bra} P. BRANQUART, J. LEWI & J.P. CARDINAEL "A context-free syntax of ALGOL 68". T.N.66 M.B.L.E. Brussels august 1970.

{Brz} J.A. BRZOZOWSKI & E.J. McCLUSKEY "Signal flow graph techniques for sequential circuit state diagrams". IEEE Trans. EC-12, 2 (april 1963) p. 67-76.

{Cho} N. CHOMSKY "Formal Properties of Grammars".
Handbook of Mathematical Psychology p. 401-410 Vol. 2
Wiley, New York 1963.

{Fos1} J.M. FOSTER "A syntax improving program".
The Computer Journal. Vol. 11 p. 31 1968.

{Fos2} J.M. FOSTER "Automatic syntactic analysis".
McDonald/Elsevier Computer Monographs London/New York 1970.

{Gin1} S. GINSBURG & H.G. RICE "Two families of languages related to ALGOL".
J.A.C.M. Vol. 10 p. 350-371 1962.

{Gin2} S. GINSBURG & G.F. ROSE "Some recursively unsolvable problems in ALGOL-like languages".
J.A.C.M. Vol. 10 p. 29-47 1963.

{Gin3} S. GINSBURG & G.F. ROSE "Operations which preserve definability in languages".
J.A.C.M. Vol. 10 p. 175-195 1963.

{Gin4} S. GINSBURG & S. GREIBACH "Deterministic Context-Free Languages".
I.C. Vol. 9 p. 620-648 1966.

{Gre1} S. GREIBACH "Formal parsing systems".
C.A.C.M. Vol. 7 p. 499-504 1964.

{Gre2} S. GREIBACH "A new normal form theorem for context-free phrase structure grammars".
J.A.C.M. Vol. 12 p. 42-52 1965.

{Gri1} M. GRIFFITHS & M. PELTIER "Self-contained ALGOL List-Pack".
Note technique I.M.A.G. février 1968.

{Gri2} M. GRIFFITHS & M. PELTIER "Grammar Transformation as an aid to Compiler Production".
Note technique I.M.A.G. février 1968.

{Gri3} M. GRIFFITHS "How the use the Grammar Transformer".
Note technique I.M.A.G. octobre 1968.

{Gri4} M. GRIFFITHS "Analyse déterministe et compilateurs".
Thèse. Grenoble octobre 1969.

{Gri5} M. GRIFFITHS & M. PELTIER "A Macro-generable language for the 360 computer".
The Computer Bulletin Vol. 13 p. 389-590 1969.

{Grif} T.V. GRIFFITHS & S.R. PETRICK "On the relative efficiencies of context-free grammars recognizers".
C.A.C.M. Vol. 8 p. 289 1965.

{Hop} J.E. HOPCROFT & J.D. ULLMAN "Formal languages and their relation to automata".
Addison-Wesley 1969.

{Knu1} D.E. KNUTH "On the translation of languages from left to right".
I.C. Vol. 8 p. 607-639 1965.

{Knu2} D.E. KNUTH "Top down syntax analysis".
International Summer School on Computer Programming, Copenhagen august 1967.

{Kor} A. KORENJAK & J. HOPCROFT "Simple deterministic languages".
Proceedings 7th Symposium on Switching and Automata Theory IEEE p. 36 1966.

{Kur1} R. KURKI-SUONIO "On top to bottom recognition and left recursion".
C.A.C.M. july 1966.

{Kur2} R. KURKI-SUONIO "Notes on top-down languages".
BIT Vol. 9 p. 225-238 1969.

{Len} A. LENTIN & M. GROSS "Notions sur les grammaires formelles".
Gauthier-Villars PARIS 1967.

{Lew} P.M. LEWIS & R.E. STEARNS "Syntax directed transduction".
J.A.C.M. Vol. 15 p. 465-488 july 1968.

{Loe} J. LOECKX "An algorithm for the construction of bounded-context parsers".
C.A.C.M. Vol. 13-5 p. 297-307 may 1970.

{Mar} F. MARTIN "Détermination de certaines caractéristiques des grammaires et langages context-free".
Thèse. GRENOBLE mai 1969.

{McN} R. McNAUGHTON & H. YAMADA "Regular expressions and state graphs for automata".
IRE Transactions on electronic computers EC-9, p. 39-47 1960.

{Pau} M.C. PAULL & S.H. UNGER "Structural equivalence and LL-k grammars".
Proceedings 9th Symposium on Switching and Automata Theory
p. 176-186 october 1968.

{Pic} E. PICHAT "Contribution à l'algorithmique non-numérique dans les ensembles ordonnés".
Thèse. GRENOBLE octobre 1970.

{Ros1} D.J. ROSENKRANTZ "Matrix equations and normal form theorem for context-free grammars".
J.A.C.M. Vol. 14 p. 501-507 1967.

{Ros2} D.J. ROSENKRANTZ & R.E. STEARNS "Properties of deterministic top-down grammars".
ACM Symposium on the theory of computing Mariva del Rey, California, may 5, 1969.

{Sal} A. SALOMAA "Theory of automata".
Pergamon Press 1969.

{Sch} M.P. SCHUTZENBERBER & N. CHOMSKY "The algebric theory of context-free languages".
Computer programming and formal systems Amsterdam North Holland
p. 118-161 1963.

{Sim} M. SIMONET "Une grammaire context-free d'ALGOL 68".
AFCET Congrès d'Informatique. PARIS 21-25 septembre 1970.

{Ter} G. TERRINE Rapport de Projet de D.E.A.
GRENOBLE 1969.

{Tix} V. TIXIER "Recursive functions of regular expressions in language analysis".
TR cs58 Computer Science Dept. Stanford California march 1967.

{Urs} G. URSCHLER "Concrete syntax of PL/1".
IBM Laboratory VIENNA TR.25.096 june 1969.

{War} S. WARSHALL "A theorem on boolean matrices".
J.A.C.M. p. 11-12 janvier 1962.

{Wo1} D. WOOD "The normal form theorem : another proof".
The Computer Journal Vol. 12 p. 139-147 1969.

{Wo2} D. WOOD "The theory of left factored languages".
part I & II. The Computer Journal Vol. 12-4, p. 349-356 1969.
& Vol. 13-1, p. 55-62 1970.

{Wo3} D. WOOD "A generalised normal form theorem for context-free grammars".
The Computer Journal Vol. 13-3 p. 272-277 1970.

{Wo4} D. WOOD "A note on top-down deterministic languages".
BIT Vol. 9 p. 387-399 1969.

{Wo5} D. WOOD "A further note on top-down deterministic languages".
Courant Institute of Mathematical Sciences New York University
july 1970.

{Wow} P.M. WOODWARD "A note on Foster's syntax improving device".
R.R.E. Memorandum n° 2352, Royal Radar Establishment, Malvern
1966.

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