Outils transportables d'aide à la réalisation d'un compilateur de langage de commande réseau

Kadria El Sanhoury

THESE

présentée à

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR INGENIEUR
INFORMATIQUE

par

Kadria EL SANHOURY

OUTILS TRANSPORTABLES D'AIDE A LA REALISATION D'UN COMPILATEUR DE LANGAGE DE COMMANDE RESEAU

Thèse soutenue le 19 novembre 1976 devant la Commission d'Examen

Président : L. BOLLIET

Examinateurs : M. BERTHAUD, P. JORRAND, F. PECCOUD

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : M. Philippe TRAYNARD

Vice-Président : M. Pierre-Jean LAURENT

PROFESSEURS TITULAIRES

MM. BENOIT Jean	Radioélectricité
BESSON Jean	Electrochimie
BLOCH Daniel	Physique du solide
BONNETAIN Lucien	Chimie Minérale
BONNIER Etienne	Electrochimie et Electrométallurgie
BOUDOURIS Georges	Radioélectricité
BRISSONNEAU Pierre	Physique du solide
BUYLE-BODIN Maurice	Electronique
COUMES André	Radioélectricité
DURAND Francis	Métallurgie
FELICI Noël	Electrostatique
FOULARD Claude	Automatique
LESPINARD Georges	Mécanique
MOREAU René	Mécanique
PARIAUD Jean-Charles	Chimie-Physique
PAUTHENET René	Physique du solide
PERRET René	Servomécanismes
POLOUJADOFF Michel	Electrotechnique
SILBER Robert	Mécanique des Fluides

PROFESSEUR ASSOCIE

M. ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM. BLIMAN Samuel	Electronique
BOUVARD Maurice	Génie Mécanique
COHEN Joseph	Electrotechnique
LACOUME Jean-Louis	Géophysique
LANCIA Roland	Electronique
ROBERT François	Analyse numérique
VEILLON Gérard	Informatique Fondamentale et Appliquée
ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM. ANCEAU François	Mathématiques Appliquées
CHARTIER Germain	Electronique
GUYOT Pierre	Chimie Minérale
IVANES Marcel	Electrotechnique
JOUBERT Jean-Claude	Physique du solide
MORET Roger	Electrotechnique Nucléaire
PIERRARD Jean-Marie	Mécanique
SABONNADIERE Jean-Claude	Informatique Fondamentale et Appliquée
Mme SAUCIER Gabrièle	Informatique Fondamentale et Appliquée

MAITRE DE CONFERENCES ASSOCIE

M. LANDAU Ioan	Automatique

CHERCHEURS DU C.N.R.S. (Directeur et Maître de Recherche)

MM. FRUCHART Robert	Directeur de Recherche
ANSARA Ibrahim	Maître de Recherche
CARRE René	Maître de Recherche
DRIOLE Jean	Maître de Recherche
MATHIEU Jean-Claude	Maître de Recherche
MUNIER Jacques	Maître de Recherche

Je tiens à rendre un sincère hommage à Monsieur Jean DU MASLE qui a été à l'origine de ce travail.

Ses conseils, ses qualités humaines et sa haute compétence m'ont été très précieux.

Il n'a jamais épargné son temps pour me diriger et m'encourager dans ce travail et son décès subit a été pour moi une grande peine.

Le travail présenté dans cette thèse a été effectué à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble.

Je tiens à exprimer ma respectueuse reconnaissance et mes remerciements à Monsieur le Professeur Louis BOLLIET pour la bienveillante attention qu'il m'a toujours accordée et pour m'avoir fait l'honneur de présider le jury de cette thèse,

Je remercie sincèrement Monsieur Philippe JORRAND pour l'intérêt qu'il a porté à mon travail ; son aide, ses conseils et ses encouragements ont été pour moi un soutien précieux ; qu'il veuille trouver ici l'expression de ma reconnaissance,

Je remercie sincèrement Monsieur Michel BERTHAUD qui a bien voulu s'intéresser à mon travail et accepter de faire partie du jury,

Mes remerciements sincères et ma reconnaissance à Monsieur François PECCOUD pour son aide amicale, ses encouragements et pour les nombreux conseils qu'il m'a donnés pendant la rédaction de ce travail, et pour sa participation à mon jury,

Je remercie dans une même pensée amicale mes collègues du Centre Interuniversitaire de Calcul de Grenoble et mes amis, pour toutes les formes d'aide qu'ils m'ont apportées.

Je remercie vivement Madame CHALAND pour sa gentillesse et sa contribution à la réalisation de ce travail avec beaucoup de sympathie.

Enfin, je remercie sincèrement Mademoiselle Elisabeth NAUDIN pour le soin, la grande compétence et la gentillesse avec lesquels elle a dactylographié ce travail, ainsi que le Service de reprographie qui a assuré la réalisation matérielle de cet ouvrage.

A la mémoire de mon Père

et à ma Mère.

TABLE DES MATIERES

CHAPITRE 4

CHAPITRE 5

CHAPITRE 6

CHAPITRE 1

INTRODUCTION.

INTRODUCTION

Le terme "réseau d'ordinateurs" est utilisé pour définir un ensemble d'ordinateurs interconnectés par des voies de télécommunication pour une mise en commun des ressources dont ils disposent.

Un des buts essentiels d'un réseau d'ordinateurs est d'offrir une grande variété de services à une communauté très large d'utilisateurs par l'intermédiaire des différents centres participant au réseau.

Parmi les projets de réseau d'ordinateurs, nous nous intéresserons particulièrement à deux exemples :

- le réseau SOC (D1, Z1)
- le réseau CYCLADES (C3).

1.a - Présentation rapide du réseau CYCLADES

CYCLADES est un projet de réseau général d'ordinateurs qui est assuré par :

- la Délégation à l'Informatique
- l'IRIA (Institut de Recherche d'Informatique et d'Automatique)
- les P et T
- les différents centres participants contribuant à la réalisation technique.

Ce projet est destiné à l'étude expérimentale du fonctionnement, de l'utilisation et de l'exploitation d'un réseau général.

Pour ce faire, le réseau CYCLADES (C5) s'est fixé les buts essentiels suivants :

- faciliter l'accès aux bases de données et les échanges d'information entre les différents points du réseau.
- offrir une grande variété de services aux utilisateurs.
- permettre la réalisation d'applications nouvelles qui ne seraient pas concevables sur des ordinateurs isolés.

CYCLADES est un réseau hétérogène reliant des machines de différentes marques (C4) :

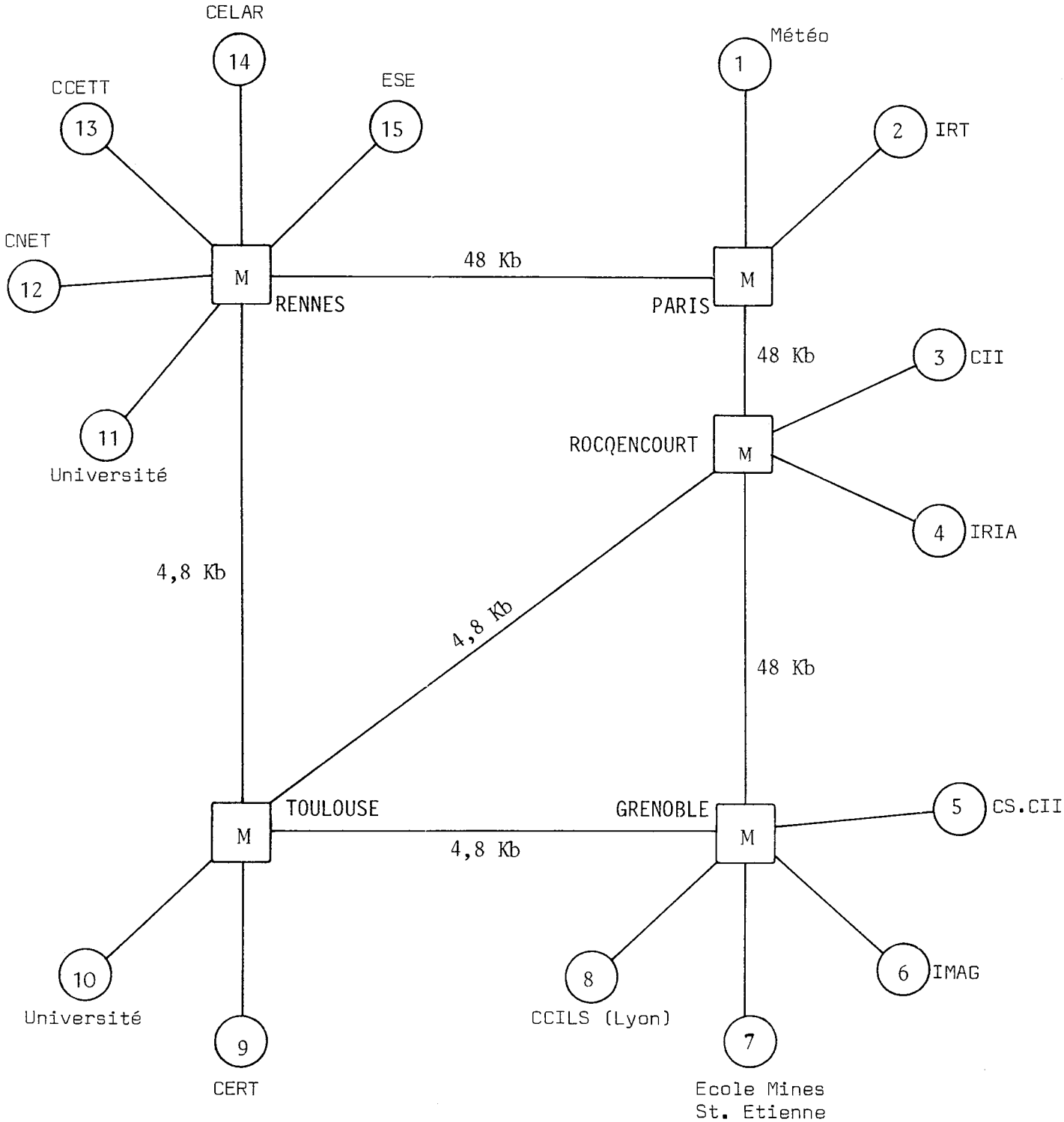
- CII IRIS 50, 80, CII 10070, CII MITRA 15
- IBM 360/67
- CDC 6600
- PHILIPS P1175
- Télémécanique T1600.

La commutation des messages est assurée par un "sous réseau" appelé CIGALE (C4, C6) formé de mini-ordinateurs CII MITRA 15 (noeuds) géographiquement répartis (Fig. 1).

Ces noeuds sont reliés entre eux par des lignes téléphoniques.

Chaque MITRA 15 noeud est également relié à un ou plusieurs centres participants par des lignes téléphoniques.

RESEAU CYCLADES



1. CDC 660	6. IBM 360/67	11. CII 10070
2. CII IRIS 80	7. Philips 1100 et T.1600	12. CII 10070
3. CII 10070 ou IRIS 80	8. CII IRIS 80	13. CII IRIS 80
4. CII 10070 et IRIS 50	9. CII 10070 et 10020	14. CII 10070
5. CII 10070	10. CII IRIS 80	15. CII IRIS 45 et HP.2100

Figure 1

1.b - Comment situer le langage de commande dans le réseau

Dans un réseau d'ordinateurs, l'utilisateur ne s'adresse pas à un ordinateur donné, mais s'adresse au réseau par l'intermédiaire d'un langage de contrôle et de commande réseau (C1).

Le but essentiel de ce langage de commande est d'offrir à l'utilisateur des outils convenables pour pouvoir spécifier au système-réseau l'utilisation qu'on veut en faire (D4 , D5).

Dans un réseau d'ordinateurs, le langage de commande réseau est un langage commun à tous les ordinateurs connectés, permettant à un utilisateur l'accès à toutes les ressources du réseau.

De même, par l'intermédiaire du langage de commande réseau, on peut s'adresser au système réseau tout entier. De plus, ce langage de commande permet la communication entre plusieurs systèmes d'exploitation différents, et, par conséquent, l'utilisation simultanée de tous les ordinateurs du réseau. En résumé, ce langage de commande réseau offre la possibilité d'exprimer dans un langage de haut niveau toutes ces opérations.

Parmi les expériences dans le domaine du langage de commande

réseau, on peut citer :

1 - LE LANGAGE EXTERNE "LE" DU RESEAU SOC (réseau homogène) (D2) :

c'est un langage de haut niveau, qui offre la possibilité de commander des opérations de type utilitaire, comme la manipulation de fichier (création, suppression, mouvement, recopie) et l'exécution de travaux. Ceci est exprimé dans le langage de contrôle du système sur lequel ces travaux doivent s'exécuter.

2 - NJCL, UN LANGAGE DE COMMANDE POUR RESEAU D'ORDINATEUR :

NJCL (L3) est un langage de programmation de haut niveau utilisable comme langage de commande dans un environnement réseau.

Ce langage permet à l'utilisateur d'indiquer les ressources indispensables à l'exécution de son travail (que ces ressources soient réparties sur un réseau d'ordinateur ou non). Ces requêtes sont exprimées sous forme de commandes introduites au moyen d'un mécanisme d'extension.

Il est également possible, par ce même procédé, de définir de nouveaux opérateurs.

Dans le cadre du réseau CYCLADES, c'est le langage de contrôle et de commande CYCLOPE qui sera offert aux utilisateurs.

Ce langage sera traduit par le compilateur en code interprétatif (le langage interne LI) (S2) ; ce langage interne sera lui-même interprété par IGOR sur l'ordinateur à qui l'utilisateur s'est adressé.

Notre travail, dans le cadre du projet CYCLADES, se situe au niveau du langage de contrôle et de commande.

1.c - La Portabilité

Comme le but de notre travail est de construire des outils logiciels transportables pour aider à la réalisation d'un compilateur de langage de commande réseau, nous ferons un bref rappel sur la portabilité des langages d'implémentation de systèmes.

Le terme portabilité signifie la possibilité de transporter des logiciels d'une machine à une autre avec le minimum d'efforts.

Dans le cadre du projet CYCLADES, réseau hétérogène reliant des machines de différentes marques, le problème de la portabilité des outils logiciels est très important.

Il est donc nécessaire d'avoir un langage bien adapté au problème considéré.

1.c.1 - Le Système FANNY

FANNY est un langage de macro-instructions. La traduction et l'expansion de ces macros ont été réalisées en deux versions :

1 - le macro-générateur et assembleur METASYMBOL est utilisé pour produire des programmes objets sur 10070.

2 - le macro-générateur STAG2 (L2) implanté sur le 10070 est utilisé pour produire des programmes sources de l'assembleur ASSIRIS pour l'IRIS 45.

Nous allons faire une présentation rapide de la structure de FANNY. Ce langage contient des macros instructions de types suivants :

- macro-instructions de chargement, qui permettent de charger le contenu d'une zone dans une autre.

- macro-instructions de comparaison et d'itération. Ces macros sont :

if - *endif* *else* - *endelse* *when* - *endwhen* *loop* - *endloop*.

- macro-instructions de manipulation de pile. Ces macros rangent ou chargent un ou plusieurs registres consécutifs dans une pile. Ces macros sont : *SAVE*, *RESTORE*.

- macro-instructions de branchement et d'appel de sous-programmes.

- macro-instructions d'addition et de soustraction

- macro-instructions de traitement de bits.

Le langage FANNY était utilisé sur les machines 10070 par l'intermédiaire du macro-assembleur METASYMBOL. Et pour que ce langage soit adapté au système IBM-360, un traducteur (macro-processeur) STAGE2 (L2) a été réalisé.

Ce macro-processeur STAGE2 est un système complètement transportable, permettant de traduire un langage de macro-instructions quelconque en langage d'assemblage d'une machine quelconque. La transportabilité est faite dans le sens IRIS-80 ou 10070 → IRIS-45 / IRIS-50 ou SIEMENS-4004 / IBM-360.

Ce système de macro-instructions FANNY ne constitue pas un langage de programmation complet. Sa structure n'est pas assez riche (problème services système d'exploitation), ni assez rigoureuse (problème calcul d'adresse). Par ailleurs, sa conception est très marquée par les règles du 10070) et les facilités de macros-génération de METASYMBOL.

Le système FANNY a néanmoins été choisi comme langage d'écriture de logiciels dans le cadre du projet CYCLADES, car il n'existait aucun autre langage convenable sur les machines du réseau. Comme exemple logiciel écrit en FANNY pour le projet CYCLADES, on peut citer la station de transport ST2 réalisée à l'IRIA.

Un autre langage transportable qui vient d'apparaître est le langage LIS.

1.c.2 - Le langage LIS

LIS, Langage d'Implémentation de Systèmes, a été créé et implémenté à la CII (L6)

C'est un langage de haut niveau dans la ligne de PASCAL.

LIS offre un système de compilation séparée intégré dans le langage. Ceci permet l'écriture de gros programmes en unités compilées séparément. Le compilateur effectue la vérification des liaisons entre unités de compilation, y compris les modules écrits en assembleur.

LIS permet aux utilisateurs de décrire des interfaces avec le système d'exploitation ou des modules écrits dans d'autres langages. Il permet également d'accéder aux caractéristiques de la machine.

Comme on l'a déjà signalé, le but de ce travail est d'obtenir des outils logiciels flexibles et transportables pour aider à la réalisation d'un compilateur de langage de contrôle et de commande, et, également, d'obtenir un outil transportable pour aider à la définition d'un langage de commande.

Au second chapitre, nous décrirons le traitement appliqué pour

varifier que la grammaire qui décrit le langage est bien LL(1), en fournissant une cross référence des renseignements nécessaires pour étudier ce langage.

Au chapitre 3, nous représentons la méthode utilisée pour l'enregistrement de la grammaire du langage en mémoire.

Au chapitre 4, nous passerons à l'analyse syntaxique de la grammaire et nous décrirons les algorithmes utilisés.

Au chapitre 5, nous décrirons les tests réalisés sur deux grammaires particulières.

Au chapitre 6, nous essaierons de tirer les conclusions et les enseignements retirés de ce travail.

CHAPITRE 2

QUELQUES OUTILS TRANSPORTABLES POUR AIDER A LA DEFINITION D'UN LANGAGE DE COMMANDE.

Le langage de contrôle et de commande utilisé par le réseau est un langage de haut niveau, défini par une grammaire LL(1).

Dans ce qui suit, nous commencerons par rappeler quelques définitions.

2.a - Rappel sur la grammaire "context-free" :

La grammaire "context-free" est un modèle pour décrire un langage, qu'il soit langage de programmation ou langage naturel.

Cette grammaire (d'après CHOMSKY) est définie par :

1 - un ensemble fini de symboles terminaux (T) dont les éléments sont dénotés a,b,c,....

2 - un ensemble fini de symboles non-terminaux (N) dont les éléments sont dénotés A,B,C,.... où les symboles terminaux et non-terminaux sont disjoints

3 - un symbole non-terminal (S) distinct dénoté 'l'axiome'

4 - un ensemble fini de règles (P) de la forme :

$$A \rightarrow \psi \quad \text{où} \quad A \in N \; \psi \in (T \cup N)^*$$

où toutes les dérivations d'une règle produisent soit une chaîne vide, soit une chaîne de symboles terminaux.

Alors, un langage L décrit par une grammaire "CF" est défini par le quadruplet (N,T,S,P).

Exemple -

Considérons un langage L décrit par la grammaire G suivante :

< S > → a < A > b
| < B > c
| d
< A > → e f
| g < B >
< B > → h
| ε

où S → est l'axiome

A,B → sont les non-terminaux

a,b,c,d,e,f,g,h → sont les terminaux

ε → est la chaîne vide.

2.a.1 - Rappel sur la grammaire LL(1) de KNUTH

Selon GRIFFITHS, la définition de KNUTH (K1), pour qu'une grammaire soit LL(1), il faut et il suffit qu'elle vérifie les quatre conditions suivantes :

1 - La grammaire ne comporte pas de récursivité à gauche, en tenant compte d'expansions éventuelles des non-terminaux qui débutent les alternatives.

Exemple -

Considérons la règle suivante, qui est récursive à gauche,

$$< B > \rightarrow a \mid < B > a$$

2 - Toutes les alternatives d'une règle doivent débuter avec des symboles terminaux différents, après expansion des non-terminaux qui débutent éventuellement ces alternatives.

Exemple -

Considérons les règles suivantes :

$$\begin{aligned} 1 - \quad & < A > \rightarrow a < B > \\ & \quad \mid b \\ & \quad \mid < B > \\ & < B > \rightarrow \ c\ d \\ 2 - \quad & < A > \rightarrow c\ b \mid < B > \\ & < B > \rightarrow \ a \\ & \quad \mid c \end{aligned}$$

le premier exemple vérifie la deuxième condition car a,b,c sont des terminaux différents, mais le deuxième exemple ne vérifie pas la condition car il y a deux alternatives qui commencent avec le même symbole c.

3 - Si d'un non-terminal A on peut dériver, d'une part, le vide et d'autre part, une chaîne $a\gamma$, et si d'un non-terminal B on peut dériver les chaînes α A β et $b\delta$, alors on a $a \neq b$.

Exemple -

Considérons les règles suivantes :

< A > → a < C >
| b
| ε
< B > → < A > c
| < C > d
< C > → c

Si on fait l'expansion de A et C, on obtient

< B > → a c c
| b c
| c
| c d

Alors, il y a deux alternatives de la règle B qui commencent par le même symbole c. Car dans l'alternative < A > c, il y a la règle A qui génère la chaîne vide, puis elle est suivie par le symbole terminal c qui appartient à l'ensemble des éléments premiers de la règle B.

Cet exemple ne vérifie pas la troisième condition.

4 - Pour chaque règle, il ne faut pas avoir plus d'une alternative qui génère la chaîne vide.

Exemple -

Considérons les règles suivantes :

1 - < A > → a < B >
| < C >
| ε
< C > → b
| ε

2 - < A > → a < B >
| < D > b
< D > → C
| ε

Pour le premier exemple, il y a deux alternatives de A qui génèrent la chaîne vide, alors elle ne vérifie pas cette condition, mais le deuxième exemple vérifie cette condition.

2.b - Comment vérifier que cette grammaire est bien LL(1)

Une grammaire de type LL(1) définit une classe de langage dont l'analyse syntaxique peut être faite de manière déterministe. Le but de cette étape est d'obtenir un outil de développement qui soit transportable pour aider à la définition du langage de commande et la construction de la représentation interne de la grammaire.

On signale le travail remarquable de Griffiths et Peltier "Transformateur de Grammaire (G4) dont un des buts est de vérifier les conditions LL(1). Ce travail est fait en utilisant le langage PL/1.

Comme notre but est d'avoir un outil transportable nous utilisons le langage ALGOL W, implanté sur plusieurs ordinateurs, pour réaliser cette étape. Le langage ALGOL W est relativement plus indépendant de la machine que ne l'est PL/1, la transportabilité d'un programme ALGOL W est meilleure.

Pour notre grammaire, nous poserons comme condition que toutes les règles de grammaire et ses alternatives commencent par un symbole terminal. En effet, il s'agit d'un langage de commande où chaque commande commence par un symbole qui indique sa nature. Mais, d'autre part, nous acceptons l'alternative vide.

Par l'analyse de chaque règle, on construit une table qui contient le début (premier symbole) de chaque alternative de cette règle. On fait la comparaison entre ces symboles.

L'ensemble de ces symboles doit être disjoint pour pouvoir vérifier les conditions (2,4) de KNUTH.

Exemple 1 -

Considérons la règle suivante :

< A > → a < B >
| b c
| c

Cet exemple vérifie les conditions car chaque alternative débute avec un élément différent. Mais s'il y a plus d'une alternative qui commence avec le même symbole terminal, comme dans l'exemple 2 suivant, le programme sortira le message d'erreur correspondant.

Exemple 2 -

Considérons la règle suivante :

< A > → a < B > b
| b c
| a

Cet exemple ne vérifie pas les conditions.

Pour la vérification de la première condition de KNUTH (récursivité à gauche), on teste si le premier symbole de chaque règle et ses alternatives, est un non-terminal, ensuite on le compare avec le nom de cette règle pour vérifier cette condition.

Si le premier symbole de la partie droite est le même que le nom de cette règle, le programme sort un message indiquant que cette règle est récursive à gauche.

Exemple -

Considérons la règle suivante :

< A > → < A > a
| b

Cette règle est récursive à gauche.

En plus, on teste l'existence de l'axiome de cette grammaire qui sert à déterminer le point de départ de l'analyse syntaxique.

Pendant l'analyse de chaque règle de grammaire, on indique l'existence de son nom dans la partie gauche par un octet à 1 (ou à 0 si non), et pour son existence dans la partie droite de la règle avec un autre octet à 1 (ou à 0 si non).

A la fin de l'analyse, on peut distinguer l'axiome ainsi que les métaterminaux, c'est-à-dire des éléments de la grammaire qui correspondent aux notions d'identificateur, d'entier, de chaîne de caractères.

Exemple -

Prenons la grammaire suivante :

< S > → a < B > b
| C
< B > → d < C >
| e f
< C > → g h
| a
| < Rien >

Alors on atteint :

nom de règle	gauche(L)	droite(R)	
S	1	0	Axiome
B	1	1	
C	1	1	
Rien	0	1	métaterminal

2.c - Production d'une Cross Reference (XREFF) pour la grammaire

Après avoir analysé la grammaire, le programme produit une cross reference, qui permet d'étudier facilement l'écriture et les caractéristiques de cette grammaire à tester.

2.c.1 - Une description de la grammaire à tester

A partir de la grammaire de définition du langage sous forme normale de Backus (B.N.F.), on analyse l'écriture du texte de cette grammaire. Après l'analyse, le programme sort une liste de la grammaire sous la forme (B.N.F.).

Chaque règle est précédée d'un numéro correspondant à son ordre dans la grammaire.

De même, la règle qui contient des erreurs est suivie par les messages d'erreur correspondants. Une erreur peut être une erreur de l'écriture ou de l'analyse.

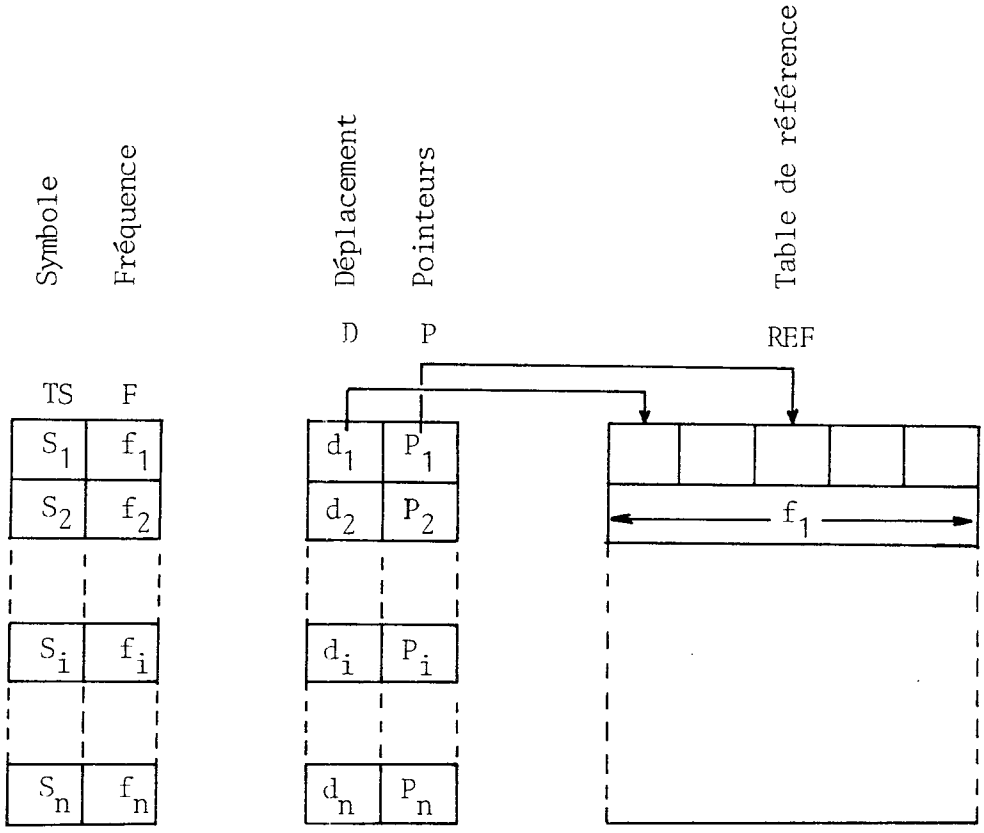
Il est très utile d'avoir une référence qui détermine les terminaux (mot-clefs), les non-terminaux, les méta-terminaux et les fonctions sémantiques qui forment cette grammaire.

A partir de la grammaire donnée sous la forme B.N.F., nous faisons l'extraction et la distinction de l'ensemble des chaînes de caractères (terminaux, non-terminaux, méta-terminaux, et les fonctions sémantiques). Puis on construit les tables de références correspondantes.

2.c.2 - La table de terminaux

C'est une référence qui détermine le nom de chaque terminal et sa fonction sémantique, aussi une table de références qui indique les numéros des règles dans lesquelles ce terminal est trouvé.

La méthode permettant d'obtenir cette référence est représentée par le schéma suivant :



où

TS → est la table des symboles (terminaux) et ses éléments sont dénotés par $s_1, s_2, \ldots s_i, \ldots s_n$.

F → est la table des fréquences de chaque terminal dans la grammaire, ses éléments sont : $f_1, f_2, \ldots f_i, \ldots f_n$.

n → nombre maximal de symboles.

REF → est la table générale des références qui contient les numéros des règles dans lesquelles ces symboles (terminaux) sont trouvés.
la dimension de cette table est $\sum_{i=1}^{n} f_i$.

D → chaque élément de cette table présente le début de la table de référence de ce terminal dans la table générale des références (REF).

Les éléments de cette table sont dénotés $d_1, d_2, \ldots d_n$.

Pour $i=1$ $d_1=1$ où $1 \leq i \leq n$

Pour un élément $i+1$, $d_{i+1} = d_i + f_i$

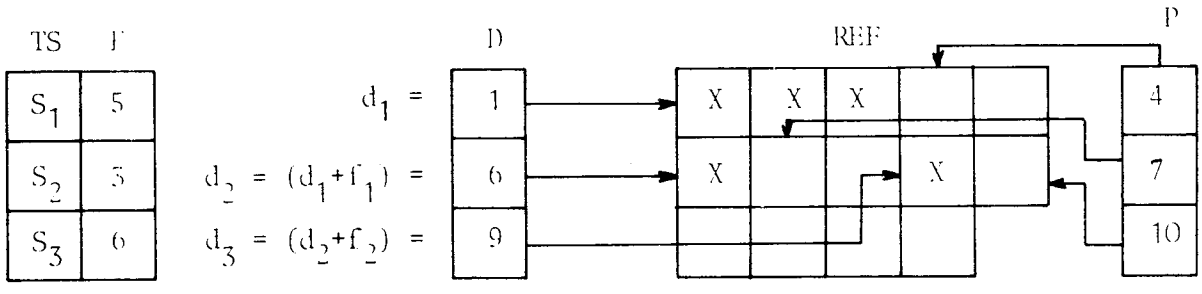
P → est un pointeur de déplacement de la table de référence de ce symbole (terminal) et ses éléments sont dénotés par $p_1, p_2, \ldots p_i, \ldots p_n$ où $d_i \leq p_i \leq f_i$, $1 \leq i \leq n$.

Exemple -

Dans cet exemple nous allons illustrer cette méthode permettant d'obtenir les références.

Prenons les symboles s_1, s_2, s_3 avec leurs fréquences dans la grammaire $f_1 = 5$, $f_2 = 3$, $f_3 = 6$.

Alors, nous avons les tables suivantes :



Quand on rencontre le symbole s_1 pendant le parcours dans la grammaire, on met le numéro de la règle dans laquelle ce symbole est trouvé dans l'élément qui a l'indice P_1 de la table de référence. Puis on avance P_1 par $p_1 = p_1+1$ pour la prochaine rencontre de ce symbole dans la grammaire.

Le même algorithme est utilisé pour s_2, s_3. Dans le schéma précédent on trouve que $p_1 = 4$, c'est-à-dire, on a déjà rencontré le symbole s_1 trois fois pendant le parcours dans le texte.

Maintenant, prenons un exemple qui indique la référence dans le cas de terminaux.

<u>Exemple</u> -

Soit les règles suivantes :

1 $< A > \rightarrow a < B > b$

$\quad | \; b \; c$

2 $< B > \rightarrow d < C >$

$\quad | \; c < C >$

$\quad | \; e$

3 $< C > \rightarrow f \; c$

$\quad | \; a$

$\quad | \; \varepsilon$

On peut associer explicitement des fonctions sémantiques aux symboles terminaux. Cependant, si on le le fait pas, les noms suivants sont produits et peuvent servir à la suite pour attacher ultérieurement des fonctions sémantiques.

Alors on obtient les références :

Terminal	Fonction sémantique	Référence
a	F1	1, 3
b	F2	1, 1
c	F3	1, 2, 3
d	F4	2
e	F5	2
f	F6	2
N_T max = 6		

à la fin le nombre maximal des terminaux est indiqué.

2.c.3 - La table des fonctions sémantiques

C'est une référence qui indique le nom de chaque fonction et les numéros des règles dans lesquelles elle est apparue.

Pour obtenir cette référence, on suit le même algorithme utilisé dans le cas des terminaux.

Exemple -

Pour les règles suivantes :

1 $< A > \rightarrow a \not\in F1\ b \not\in F2 < B >$

$\quad | c \not\in F3$

2 $< B > | d \not\in F4\ e \not\in F5$

$\quad | f \not\in F6 < C >$

3 $< C > \rightarrow b \not\in F2\ a \not\in F1$

$\quad | e \not\in F5$

$\quad | \varepsilon$

on obtient :

Fonction	Références
F1	1, 3
F2	1, 3
F3	1
F4	2
F5	2, 3
F6	2
N_F max = 6	

N_F max est le nombre maximal de fonctions dans la grammaire.

2.c.4 - La table des non-terminaux

Cette référence est composée de trois parties :

1 - une table détermine le nom de chaque non-terminal.

2 - une table de référence qui indique les numéros des règles dans lesquelles ce non-terminal est apparu.

3 - un commentaire qui indique le type de ce symbole (axiome ou méta-terminal).

A la fin, le nombre maximal des non-terminaux dans cette grammaire est indiqué.

On utilise le même algorithme que dans le cas des terminaux pour obtenir cette référence.

Pour les commentaires, on utilise les tables d'octets qui indiquent l'existence du non-terminal en partie gauche et l'autre octet pour l'indication de son existence dans la partie droite.

Exemple -

Examinons la grammaire suivante :

1 $< S > \rightarrow a < B > b$

$\mid c$

$\mid < ID >$

2 < B > → db < C >

| e

3 < C > → f

| < RIEN >

On obtient les références suivantes :

non-terminal	Commentaire	Référence
S	Axiome	1
B		1, 2
C		2, 3
ID	méta-term.	1
RIEN	méta-term.	3
N_R max = 5		

De même on obtient la table de métaterminaux.

Méta-term
ID
RIEN
N_M max = 2

Cette cross référence qu'on vient de décrire donne une aide à la définition du langage de commande de donnée.

Maintenant, nous passons au chapitre 3 où nous allons montrer la méthode utilisée pour l'enregistrement de la grammaire.

CHAPITRE 3

REPRESENTATION INTERNE DE LA GRAMMAIRE

(ORIENTEE VERS LES LANGAGES DE COMMANDE).

3 - LA REPRESENTATION INTERNE DE LA GRAMMAIRE

Cette représentation est orientée vers les langages de commande.

Pour pouvoir analyser une chaîne de symboles en entrée, il faut stocker les règles de la grammaire en mémoire. L'ordre dans lequel sont rangées les règles de la grammaire en mémoire joue un rôle quant à l'algorithme d'analyse employé (B4).

Le but de la représentation interne de la grammaire que nous allons décrire est l'enregistrement de la grammaire dans la mémoire au niveau de l'octet, qui sera utilisable par l'analyse syntaxique qui utilise un algorithme d'analyse descendante (G1).

Il y a deux facteurs importants qui jouent dans l'implémentation de la grammaire :

1 - le coût total en mémoire (l'encombrement mémoire de l'ordinateur). Dans notre cas c'est le facteur qui importe le plus.

2 - la rapidité de l'algorithme d'analyse qui sera employé.

3.a - Les méthodes d'enregistrement de grammaire existantes et leur occupation mémoire

Parmi les méthodes d'enregistrement existantes, on peut citer :

A - La méthode de précédence de Floyd (F1). Cette méthode consiste à enregistrer la grammaire et une matrice de précédence en mémoire.

Cette matrice comprend les relations entre les symboles terminaux.

En appliquant cette méthode sur les deux grammaires suivantes, on obtient :

1 - Pour le langage PASCAL-S, défini par une grammaire comportant 45 règles, 48 symboles terminaux et 45 symboles non-terminaux, il faut :

- une matrice de 48 × 48 relations × 2 bits → 576 octets
- la grammaire occupe 450 octets y compris les pointeurs

Dans ce cas nous arrivons à 1026 octets.

2 - Pour le langage externe LE de SOC défini par une grammaire comportant
47 règles, 63 symboles terminaux et 47 symboles non-terminaux, il faut :

- une matrice de 63 × 63 relations × 2 bits → 992 octets
- 600 octets pour la grammaire et ses pointeurs.

Dans ce cas, il faut au total 1592 octets.

B - La méthode de Brasseur et Cohen (B2). Dans cette méthode la représentation de la grammaire est traitée comme un ensemble d'arbres. Par cette méthode la grammaire est stockée en trois tableaux :

1 - le tableau (ARBE) contient les éléments de la grammaire, avec

un symbole spécial marquant à la fin de chaque règle.

2 - le tableau (PRECEDANT) indique quel élément précède celui du tableau ARBE.

3 - le tableau (ALTERNANT) qui indique où se trouve, dans ARBE, le début de la branche suivante.

En appliquant cette méthode sur les deux grammaires déjà citées, on obtient :

- Pour la grammaire de PASCAL-S qui comporte 270 éléments, il faut 1800 octets au total.

- Pour la grammaire du langage externe qui comporte 350 éléments, il faut 2200 octets, coût total en mémoire.

Par la méthode que nous allons décrire, nous arrivons à représenter la grammaire du langage PASCAL-S en mémoire, par 676 octets.

De même, la représentation de la grammaire du langage externe occupe 1202 octets.

Dans notre méthode, les appels aux fonctions sémantique sont compris.

Cette méthode d'enregistrement de la grammaire est inspirée par une méthode de DU-MASLE (D3), (Brooker, Macallum, Morris et Rohl) (B4). Dans ce qui suit nous décrirons cette méthode d'enregistrement.

3.b - Description de la méthode d'enregistrement appliquée

Dans cette représentation, chaque symbole du vocabulaire de la grammaire, est précédé par un octet qui détermine son type comme symbole de la grammaire.

Nous considérons les trois cas suivants.

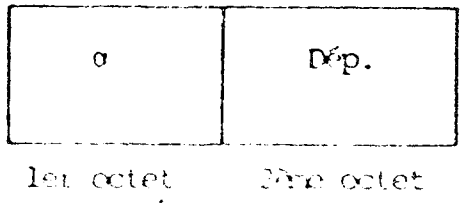
3.b.1 - Alternative d'une règle

Le début de chaque enregistrement d'une règle de la grammaire qui possède des alternatives, ou l'alternative qui est suivie par une autre, est précédé par deux octets.

(a) le premier octet indique l'existence d'une alternative. Ce symbole d'indication va être noté par α .

(b) le deuxième octet contient une adresse relative. Cette adresse est un déplacement relatif par rapport au début de l'enregistrement de cette alternative. C'est-à-dire que ce déplacement pointe sur le début de la prochaine alternative.

En pratique, ce déplacement est < 255 octets. Le schéma suivant montre cette représentation.



Exemple -

Considérons la grammaire suivante :

$$< S > \rightarrow \underset{\alpha}{\underbrace{bc}} \mid \underset{\alpha}{\underbrace{d}} < A >$$

$$< A > \rightarrow \underset{\alpha}{\underbrace{e}} \mid \underset{\alpha}{\underbrace{f}} \mid \varepsilon$$

Les flèches montrent la signification de α .

3.b.2 - Symbole Terminal

Les symboles considérés comme des symboles terminaux sont : les mots-clés et les meta-terminaux, c'est-à-dire les identificateurs, les entiers, les chaînes de caractères et la chaîne vide.

Chaque symbole terminal sera représenté par trois octets :

(a) Le premier octet contient le type de ce symbole. Dans ce cas, on va le noter par γ .

(b) Le deuxième octet contient le code de ce symbole terminal.

(c) Le troisième octet contient la fonction sémantique qui est générée automatiquement à l'occurrence d'un symbole terminal dans la grammaire, sauf à l'occurrence d'une chaîne vide.

Remarquer que la chaîne vide est représentée par deux octets/ Le schéma suivant montre cette représentation :

pour la chaîne vide :

où

γ → est le type du symbole

C → est le code du symbole terminal

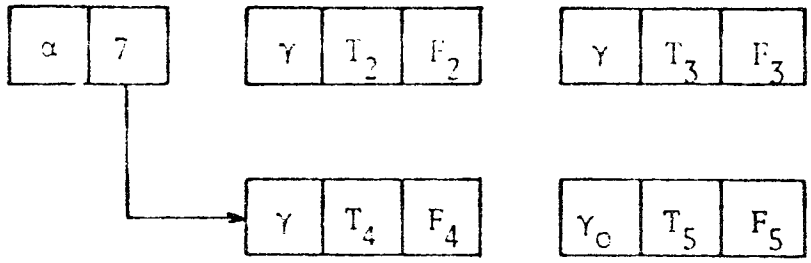
F → est la fonction sémantique.

Dans le cas d'un symbole terminal qui termine une règle ou une alternative d'une règle, on utilise le symbole γ_0 pour le distinguer.

Exemple -

Considérons la règle suivante d'une grammaire,

$$< A > \rightarrow \alpha \; T_2 \; T_3 \mid T_4 \; T_5$$



Ce schéma montre la représentation de cette règle dans la mémoire.

3.b.3 - Symbole non-terminal

Cette classe de symboles est représentée par deux octets.

(a) Noter que le premier octet est toujours réservé pour indiquer l'occurrence de ce type de symbole dans la grammaire.

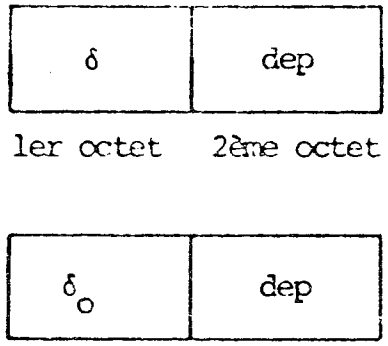
Dans ce cas, on utilise le symbole noté δ pour indiquer ce type de symbole.

(b) Le deuxième octet sert comme un pointeur vers la règle correspondante de la grammaire. Ce pointeur est un déplacement relatif par rapport au point d'appel. Ce déplacement relatif occupe un octet ou deux octets selon les cas suivants :

1. Si le déplacement relatif est inférieur à 255 octets, on utilise un octet pour cette représentation.

Remarquer que, dans le cas où le symbole non-terminal termine une règle ou une alternative d'une règle, le symbole noté δ_0 est utilisé pour distinguer ce cas.

Le schéma suivant montre la représentation de ce cas

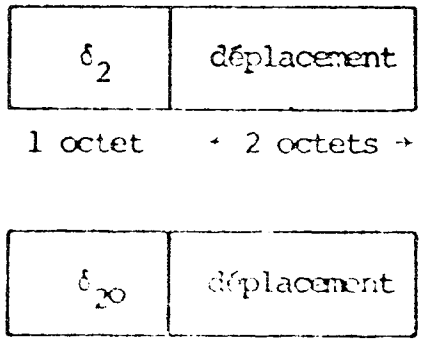


2. Si le déplacement relatif est > 255 octets, alors le pointeur occupe deux octets.

Notons qu'on utilise un autre type de symbole pour indiquer l'occurrence de ce cas. Ce symbole est noté δ_2 .

De même, on considère le symbole δ_{2^o} comme une indication de l'occurrence d'un symbole non-terminal qui termine une règle ou une alternative d'une règle.

Cette représentation est illustrée par le schéma suivant :



Remarquons qu'il faut considérer le déplacement relatif positif

(i.e. pointeur en avant) et le déplacement relatif négatif (i.e. pointeur en arrière). On va les distinguer par les symboles (+) ou (-) à côté du symbole d'indication du type. Par exemple δ_+ ou δ_-.

Dans ce qui suit, nous montrons cette méthode d'enregistrement par quelques exemples.

3.b.4 - Exemples

Les exemples suivants illustrent cette méthode d'enregistrement de la grammaire.

Considérons un langage L décrit par la grammaire G,

où

S représente l'axiome de cette grammaire

A,B,C,... représente l'ensemble des non-terminaux

$T_{(i)}$, $i = 1 \rightarrow n$ représente l'ensemble des terminaux

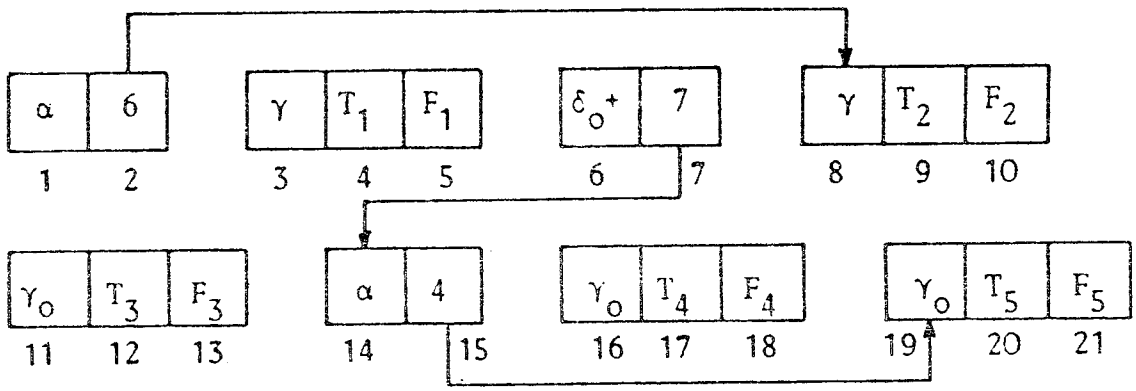
ε représente la chaîne vide.

Exemple 1 -

Prenons la grammaire suivente :

$$< S > \rightarrow T_1 < A >$$
$$\quad | \; T_2 \; T_3$$
$$< A > \rightarrow T_4$$
$$\quad | \; T_5$$

Le schéma suivant montre la méthode d'enregistrement :



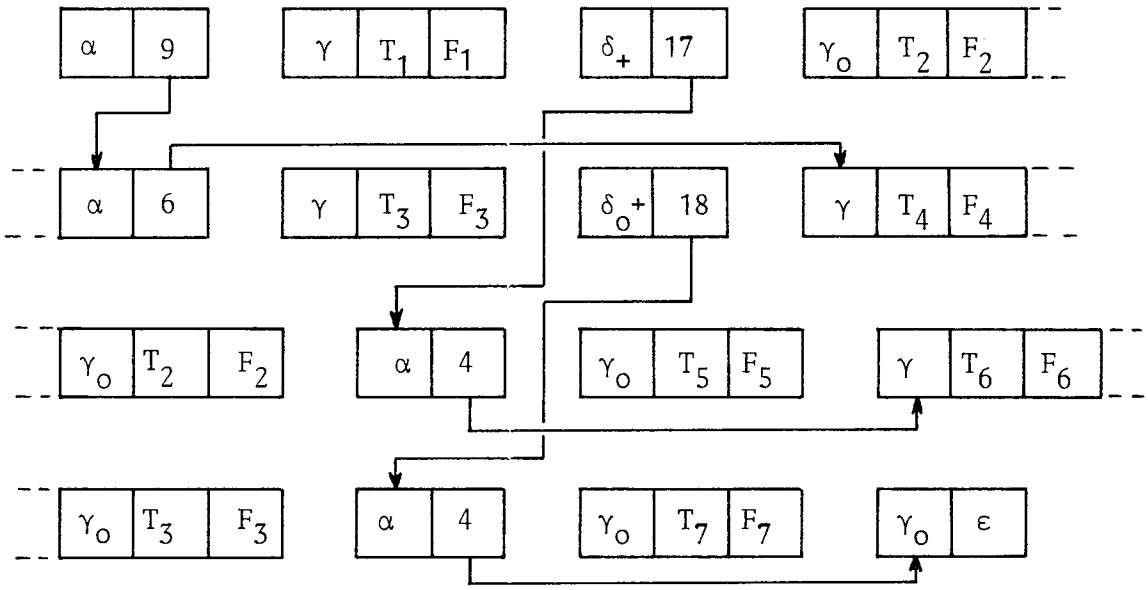
Par cet exemple nous montrons la signification de α, γ, γ_o et δ_{o+} .

Exemple 2 -

Supposons la grammaire suivante,

$$
\begin{aligned}
&< S > \rightarrow T_1 < A > T_2 \\
&\quad\quad | \; T_3 < B > \\
&\quad\quad | \; T_4 \; T_2 \\
&< A > \rightarrow T_5 \\
&\quad\quad | \; T_6 \; T_3 \\
&< B > \rightarrow T_7 \\
&\quad\quad | \; \varepsilon
\end{aligned}
$$

Par le schéma suivant on représente la méthode d'enregistrement de cette grammaire.



Cette représentation montre la signification de α, γ, γ_o, δ_+ et δ_{o+} .

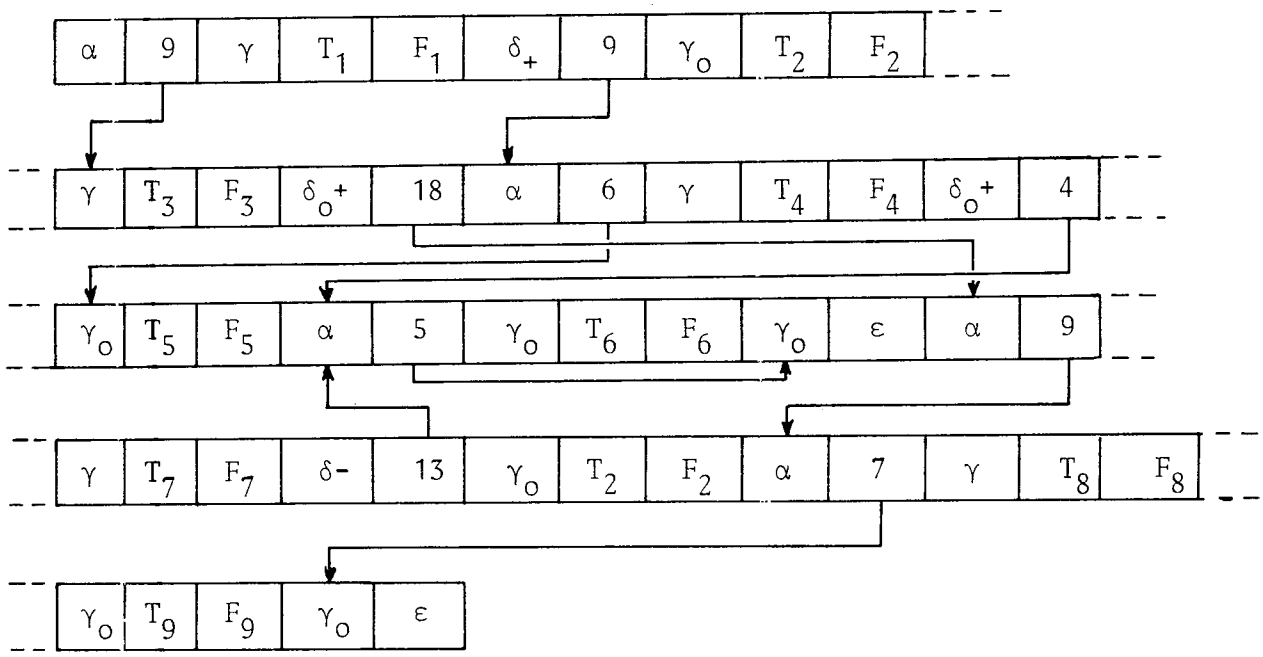
Exemple 3 -

Considérons la grammaire suivante,

$$< S > \rightarrow T_1 < A > T_2$$
$$\mid T_3 < C >$$
$$< A > \rightarrow T_4 < B >$$
$$\mid T_5$$
$$< B > \rightarrow T_6$$
$$\mid \varepsilon$$
$$< C > \rightarrow T_7 < B > T_2$$
$$\mid T_8 T_9$$
$$\mid \varepsilon$$

Le schéma suivant représente l'enregistrement de cette grammaire.

Par cet exemple, nous montrons la signification de α, γ, γ_o, δ_+, δ_{o+} et δ_- (pointeur en arrière).



3.c - Mise en pratique de l'enregistrement de la grammaire donnée

Après avoir vu cette représentation de la méthode utilisée pour l'enregistrement de la grammaire, nous décrirons en bref la méthode utilisée pour réaliser cette représentation de la grammaire.

Cette méthode est de générer une suite d'appels à des macros instructions écrites en langage transportable de macros [FANNY] (L1).

Le but de ces macros est de générer cette représentation de la grammaire dans la mémoire.

Ces macros sont :

1. la macro instruction ALTER

Cette macro a la forme suivante :

ALTER Parm (Dep)

Cette macro signifie l'occurrence d'une alternative et son paramètre formel Parm (dep) représente le déplacement relatif en octets.

2. les macros instructions TERMINAL et TERMINALT

La macro Terminal a la forme suivante :

TERMINAL $Parm_1$ (code), $Parm_2$ (fun)

Cette macro signifie l'occurrence d'un symbole terminal ou un meta-termi-

nal et ses paramètres formels $Parm_1$, $Parm_2$, représente :

- le code de ce symbole terminal
- la fonction sémantique de ce symbole.

Si le symbole terminal termine une règle ou une alternative d'une règle, la macro TERMINALT est utilisée à la place de la macro TERMINAL, car chaque macro génére un code différent (γ ou γ_o).

3. les macros instructions RULE et RULET

La macro RULE a la forme suivante :

RULE Parm(1) , Parm(2)

Cette macro signifie l'occurrence d'un symbole non-terminal.

- le premier paramètre Parm(1) contient le code δ_+ (déplacement positif) ou δ_- (déplacement négatif).
- le deuxième paramètre Parm(2) contient le déplacement relatif en octets par rapport au point d'appel.

Dans le cas où le symbole non-terminal termine une règle ou une alternative de règle, la macro RULET est utilisée à la place de la macro RULE. Dans ce cas Parm(1) contient le code (δ_{o_+} ou δ_{o_-}).

Exemple -

Pour illustrer cette représentation d'enregistrement de la grammaire en forme de macros instructions, on prend la grammaire de l'exemple 1 suivant :

$$< S > \rightarrow T_1 < B >$$
$$\quad | \; T_2 \; T_3$$
$$< B > \rightarrow T_4$$
$$\quad | \; T_5$$

Dans ce cas on doit générer la suite d'appels suivants :

Séquence	Nom de macro	Paramètre
1	ALTER	(6)
2	TERMINAL	(T_1), (F1)
3	RULET	(δ_{o^+}), (7)
4	TERMINAL	(T_2), (F2)
5	TERMINALT	(T_3), (F3)
6	ALTER	(4)
7	TERMINALT	(T_4), (F4)
8	TERMINALT	(T_5), (F5)

CHAPITRE 4

ANALYSE SYNTAXIQUE DE LA GRAMMAIRE.

4 - L'ANALYSE SYNTAXIQUE DE LA GRAMMAIRE

Le but de l'analyse syntaxique est de décider si une séquence de symboles est en accord avec les spécifications syntaxiques correspondant au langage donné ou non.

Pour faciliter le travail de l'analyseur, on convertit les éléments de cette séquence en des unités syntaxiques correspondantes.

Pour cette étape l'analyse lexicographique (B1, G3) est utilisée.

4.a - L'analyse lexicographique

Le programme utilisé pour cette analyse s'appelle l'éditeur constitué d'un sous-programme appelé à l'intérieur de la phase d'analyse syntaxique toutes les fois qu'on a besoin d'une nouvelle unité syntaxique.

Le but de l'éditeur est de transformer les images-cartes d'un programme-source de ce langage de commande en des unités syntaxiques utilisables par l'analyseur.

L'avantage de cette transformation est de faciliter le traitement fait par l'analyseur en appliquant les règles de calcul du langage sans tenir compte des règles d'écriture.

Dans notre cas, l'éditeur n'emploie pas la méthode consistant à

transformer tout d'abord le programme-source en chaîne codée avant de la faire passer élément par élément à l'analyseur (B3), mais il travaille en parallèle avec l'analyseur ; c'est-à-dire qu'après avoir formé une unité syntaxique, il passe au traitement de l'analyseur.

Cette méthode de traitement de l'éditeur permet de réaliser une réduction du nombre de mémoires utilisées par l'éditeur.

De même cela permet d'adapter plus facilement le compilateur pour un travail en mode interactif.

La méthode de l'enregistrement de la grammaire que nous employons ici donne la possibilité d'appliquer cette méthode de traitement de l'éditeur.

4.a.1 - <u>La définition d'une unité syntaxique</u>

< unité syntaxique > → < Mot-clé >
| < identificateur >
| < entier >
| < chaîne de caractères >

< Mot-clé > | < Mot particulier du langage >
| < délimiteur >

4.a.2 - Fonction de l'analyseur lexicographique

Les fonctions principales de l'éditeur sont :

1 - Lecture du texte. L'éditeur lit les images-cartes d'un programme-source du langage de commande carte par carte.

Le texte d'une carte peut se composer de différents types d'ensembles de caractères.

Ces ensembles de caractères peuvent être :

1. Mot-clé.
2. Ensemble de lettres (identificateur).
3. Ensemble de chiffres (entier).
4. Délimiteurs.

2 - Extraction de chaque ensemble de caractères, et détermination du type de son unité syntaxique. Chaque unité syntaxique est distinguée par un code (sur un octet).

3 - Rangement de cet ensemble de caractères dans la table correspondante au type de son unité syntaxique (table d'identificateurs, chaîne de caractères, et entiers), après avoir recherché son existence dans la table.

Enfin, cet ensemble de caractères sera représenté par un code numérique.

4 - Elimination des commentaires.

5 - Détection d'erreurs lexicographiques dans le texte d'un programme-source et sortie des messages d'erreurs correspondants.

A la fin de cette étape de l'analyse lexicographique on obtient une unité syntaxique représentée par :

.- un code tenant sur un octet qui indique le type (mot-clé, identificateur, chaîne de caractères, entier, délimiteur)

.- un code pour identifier l'ensemble de caractères qui est représenté par cette unité syntaxique.

Ce code peut être :

adresse dans une table, numéro d'ordre dans une liste, ou code numérique pour un mot-clé.

4.a.3 - Algorithme

La procédure suivante est une représentation simple pour une idée générale du fonctionnement de l'éditeur.

```
Procédure   EDIT ;

    Begin

        STRING (80)  CARTE;  STRING(1)  TYPE,UNTS;

        INTEGER  I ; VAL;

    Comment  carte représente une image de carte d'entrée du programme-
             source du langage de commande.
             I est l'index de position de colonne dans cette carte.

    NEXT :   Readcard  (CARTE);  I := 0;

             while  (I ¬ = 71)  and   (CARTE (I/1) ¬ ="   ")  do

          Begin

               if   CARTE (I/1) < "A"  Then

               Begin

                    if   CARTE (I/1) = """" Then

                 ┌──────────────────────────────────────────┐
                 │ Traitement et rangement de texte ;       │
                 │   TYPE := 3 ;  VAL := adresse dans la table ; │
                 └──────────────────────────────────────────┘

                    Else

          ┌──────────────────────────────────────────────────┐
          │ Délimiteur trouvé, chercher dans la table des délimiteurs ; │
          │   TYPE := 8   ;   VAL := code numérique ;        │
          └──────────────────────────────────────────────────┘

               End   Else

                         Begin

                              if   CARTE (I/1) > "Z"  Then

                    ┌──────────────────────────────────────┐
                    │ Traitement et rangement d'entier ;   │
                    │  TYPE := 4   ;   VAL = l'ordre de son │
                    │                       rangement ;    │
                    └──────────────────────────────────────┘

                              Else
```

```
Traitement pour distinguer l'identificateur et le
mot-clé. Rangement de l'identificateur dans sa table ;
  TYPE := un code correspondant  ;  VAL := code de mot-
                clé ou pointeur pour l'identificateur

                End;
              UNTS := TYPE;
              CODE :=  VAL;
        Comment   l'unité syntaxique formée est transférée à
                  l'analyse syntaxique.
              Goto  FIN;
          End;
            Goto  NEXT
FIN:    End   EDIT;
```

4.a.4 - Traitement de mots-clés

Dans le cas du langage de commande que l'on traite, les mots-clés sont réservés. C'est-à-dire qu'ils ont la forme d'un identificateur, mais ne peuvent pas être utilisés comme des identificateurs.

Pour cette raison, un mot-clé est traité comme un cas particulier des identificateurs, en cherchant dans la table des mots-clés.

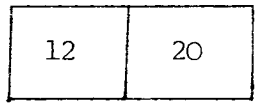
Lorsqu'on a trouvé le mot-clé en question, on le remplace par son code.

Si on ne le trouve pas dans la table, alors on le considère comme un identificateur.

Exemple -

Supposons que CPU est un mot-clé avec un code 20.

Alors ce mot-clé sera remplacé par l'unité syntaxique qui a la forme suivante :



type code

1 octet 2 octets

où le 12 représente le type de l'unité syntaxique.

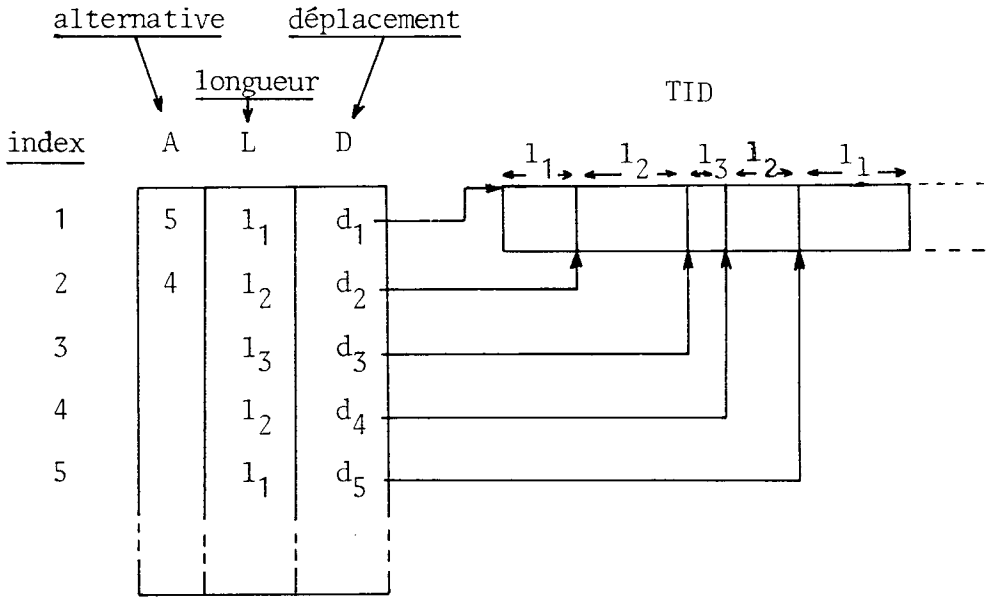
4.a.5 - Traitement d'identificateurs

L'important pour l'analyse syntaxique c'est qu'un identificateur quelconque ait été trouvé, car pour lui tous les identificateurs sont identiques.

Le but de ce traitement est de conserver les caractères formant l'identificateur dans une table et représenter l'identificateur par un code correspondant à une fonction de son index dans cette table.

En général la forme de la table utilisée dépend de la définition de l'identificateur dans le langage et de la place disponible (G3).

Dans notre cas, la longueur de l'identificateur est limitée à trente deux caractères et nous utiliserons la méthode qui est présentée par le schéma suivant :



où

TID → est la table d'identificateur.

D → est le déplacement du début de chaque identificateur.

L → est la longueur de l'identificateur.

A → est l'index de l'autre identificateur de la même longueur.

Dans cette méthode chaque identificateur peut se trouver, par sa longueur et le déplacement du début de cet identificateur, dans la table des identificateurs.

Pour éviter des recherches inutiles dans la table des longueurs, nous utiliserons un masque (fig. 2), correspondant à la longueur possible des identificateurs pour décider si cette longueur est déjà rencontrée ou non. Dans l'affirmative, on obtient l'indice du premier élément de ce groupe de longueurs donné dans la table des longueurs.

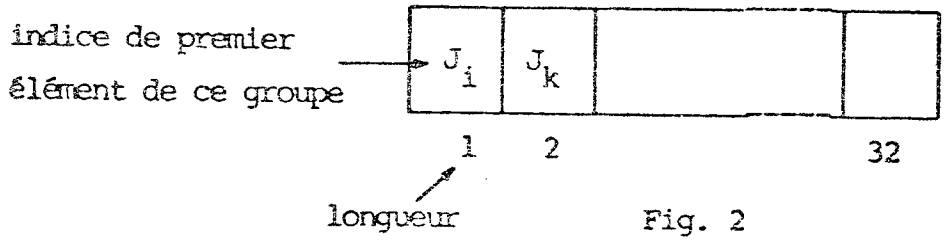


Fig. 2

Le traitement d'un nouvel identificateur est réalisé par comparaison entre cet identificateur et les identificateurs précédemment rencontrés, de même longueur.

Pour gagner du temps en cherchant un autre identificateur qui ait la même longueur, on s'adresse à la table d'alternants de longueurs qui indique directement l'index du prochain élément de la même longueur.

La procédure suivante montre cette méthode ; supposons que :

.- TID est la table des identificateurs avec M l'index du prochain élément à ranger.

.- TL est la table de longueur.

.- TA est la table d'alternants de longueur.

.- TD est la table de déplacement de début de chaque identificateur dans la table d'identificateur.

.- N est l'index du prochain élément dans ces tables.

.- MASK signifie le mask de longueur.

```
Procédure   CHERCHE;
            Begin
             Comment   ID  C'est l'identificateur en question avec une
                           longueur L ;
             if   MASK (L) = "    " then goto RANG;
               J := MASK (L);
             Comment   J index de la table TD ;
      TEST :
            Comment faire la comparaison de ID avec les identificateurs
            précédants de la même longueur. I est le début de l'identi-
            ficateur dans sa table ;
```

```
        I := TD (J);
        if   ID = TID (I/L)  then
           begin
               CODE := J; goto  FIN
           end;
        if   TA (J)   ¬= 0   then
           begin
               comment  il y un autre ident. de la même longueur ;
               J := TA (J); goto   TEST;
           end  else
                    TA (J)  := N;
    RANG :
           Comment  rangement d'un nouvel identificateur ;
           TL (N)  := L;  TD (N)  := M;
           CODE :=  N;    N := N+1;
           TID (M/L)  := ID (o/L); M := M+1;
FIN :
        END   CHERCHE;
```

4.b - Analyse syntaxique

De façon classique, l'analyse syntaxique sert à vérifier l'appartenance d'une phrase à un certain langage défini au moyen d'une grammaire. Les éléments de la phrase sont pris dans le vocabulaire de cette grammaire.

Ce processus d'analyse peut se représenter au moyen de l'arbre syntaxique dont les feuilles sont les composants de la phrase.

Un algorithme d'analyse syntaxique doit être utilisé pour réaliser cette phase. Cet algorithme dépend évidemment de la famille de langage que l'on désire traiter (B2), et un choix doit être fait pour cet algorithme.

Parmi les possibilités on trouve :

.- les algorithmes qui parcourent l'arbre syntaxique de bas en haut pour des langages de précédence (W1), LR(K) Knuth (K1).

.- Les algorithmes qui parcourent l'arbre syntaxique de haut en bas pour les langages LL(K) (F2, K1).

Ces algorithmes sont utilisés par diverses implémentations.

Dans notre cas, le langage de commande que l'on veut traiter est défini par une grammaire LL(1), et nous utilisons un algorithme d'analyse descendant qui est déterministe (G1), car l'analyse syntaxique des langages LL(1) est faite de manière déterministe (sans retour-arrière).

L'avantage de l'analyse syntaxique déterministe et descendante est de permettre l'insertion d'actions sémantiques à l'intérieur des règles de production de la grammaire.

Utilisons cette approche pour la construction automatique des compilateurs, on trouve les transformateurs de grammaire de Griffith et Peltier (G4) et du "Compiler description language" de Koster (K2).

4.b.1 - Méthode d'analyse appliquée

Le langage de commande qu'on va traiter est défini par une grammaire LL(1) dans laquelle toutes productions des règles de grammaire commencent par un symbole terminal, et tous les symboles sont différents pour toutes les alternatives d'une règle.

De plus, toutes les différentes expansions d'un symbole non-terminal doivent débuter par des symboles terminaux différents, ou, éventuellement, par une chaîne vide.

Grâce à ces caractéristiques, l'analyse est déterministe car, en comparant le premier élément d'une production, on décide sans équivoque de la validité de cette production.

L'analyse se fait en consultant l'arbre de la grammaire de haut en bas, de gauche à droite, en utilisant à chaque instant la prochaine unité syntaxique pour choisir le chemin à parcourir.

.- Pour l'analyse cn utilise une pile pour retenir l'adresse des symboles non-terminaux que l'on rencontre pendant le parcours de l'arbre syntaxique de la grammaire.

.- Chaque adresse est retenue jusqu'à ce que le symbole non-terminal correspondant soit remplacé par une production qui ne comprend que des symboles terminaux,ou bien qui est vide.

.- On enlève l'adresse du non-terminal de la pile au moment où l'on trouve sa production et on continue l'analyse.

Après avoir formé une unité syntaxique par l'éditeur, on donne le contrôle à l'analyseur.

..- L'analyse commence par la comparaison du premier élément du programme source (unité syntaxique) avec le premier élément de la première production de l'axiome (premier élément enregistré de la grammaire).

..- Dans le cas où l'unité syntaxique est conforme à l'élément de la grammaire à comparer, on fait progresser le pointeur qui pointe sur l'élément de la grammaire à comparer et on passe le contrôle à l'éditeur pour avoir une autre unité syntaxique à analyser.

..- Dans le cas d'erreurs syntaxiques, on donne le contrôle aux procédures d'erreurs qui sortent le message correspondant, et on continue l'analyse.

L'analyse d'un programme correct laisse la pile vide à la fin.

Voyons, par un exemple, le déroulement de l'algorithme.

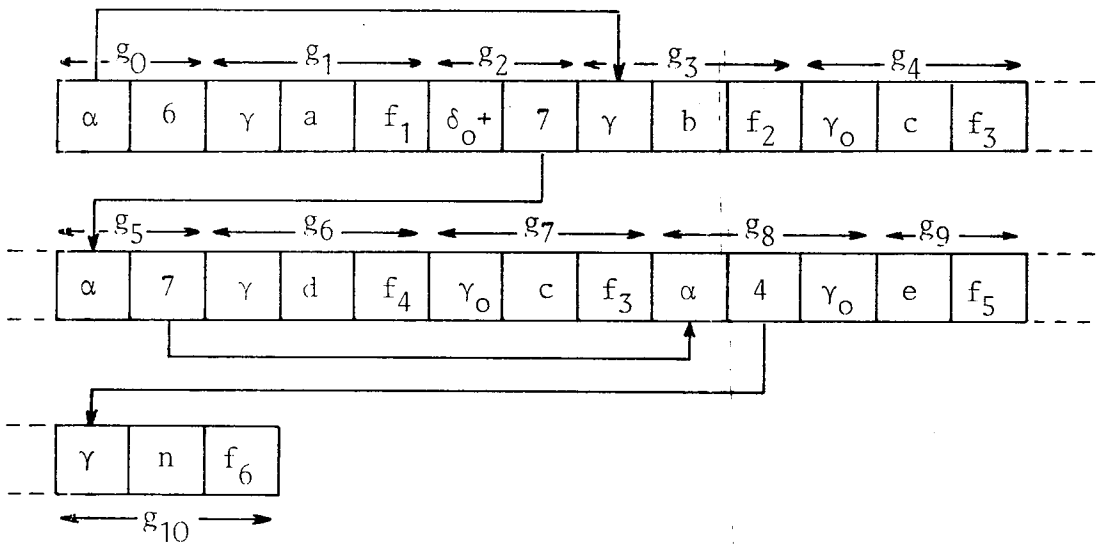
Exemple -

Soit à reconnaître la chaîne ae, étant donnée la grammaire suivante :

< S > → a < B >
| b c
< B > → d c
| e
| n

S → représente l'axiome.

La grammaire est enregistrée sous la forme :



Cette forme est déjà expliquée dans la représentation de la grammaire en mémoire où

α → signifie l'existence d'une alternative, l'octet suivant contenant le déplacement.

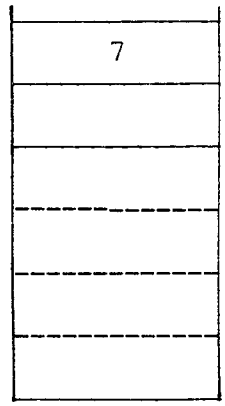
γ → signifie l'existence d'un symbole terminal.

δ → signifie l'existence d'un symbole non-terminal,

g_o, g_1, g_2,.... sont les éléments de la grammaire

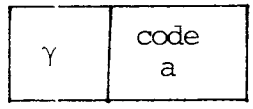
u_1, u_2,.... sont les unités syntaxiques.

la pile utilisée pour garder les adresses de non-terminaux.



..- On teste le type du premier élément g_o, si g_o est une alternative, on garde le déplacement dans un octet.

..- On avance le pointeur qui pointe sur l'élément de la grammaire à comparer, et on pointe sur g_1 . On compare le niveau de g_1 avec l'unité syntaxique u_1 qui a la forme :



Comme u_1 est équivalente à g_1, on fait progresser le pointeur d'élément de la grammaire et on pointe sur l'élément g_2.

..- On prend l'unité syntaxique suivante u_2 qui a la forme :

γ	code e

..- On teste le type de g_2 et, comme c'est un symbole non-terminal, on garde son adresse dans la pile on fait un branchement à cette règle et on remplace g_2 par g_5.

..- On teste le type de g_5, c'est une alternative, alors on garde le déplacement et on pointe sur le premier élément de la règle (g_6) et on le compare avec u_2. Comme u_2 n'est pas équivalent à g_6, on pointe sur g_8.

..- g_8 est une alternative, alors on garde le déplacement et on pointe sur g_9.

..- On compare g_9 avec le niveau de u_2, on arrive à une équivalence. On teste le type de g_9. S'il s'agit d'un élément terminal, elle n'a pas de successeur.

Si oui, comme c'est le cas de notre exemple, on retourne au symbole non terminal où l'on a quitté la règle en dernière instance par utilisation de l'adresse de ce symbole que l'on a gardé dans la pile (g_2), on enlève l'adresse de la pile.

..- On teste le type de g_2 : est-ce un élément terminal ? Et comme c'est le cas dans cet exemple, alors on arrive à trouver une production complète de la règle de la grammaire.

Alors, cette chaîne est correcte.

4.c - Suite de l'analyse syntaxique

Le rôle de l'analyse syntaxique est de consulter la grammaire prédéfinie pour déterminer si l'unité syntaxique en traitement est conforme aux règles de la grammaire.

Si l'unité syntaxique est acceptée, l'analyseur obtient de la grammaire le numéro de la fonction sémantique appropriée.

L'analyseur passe ce numéro à la procédure GENCODE où on peut faire l'appel de la fonction sémantique correspondant à ce numéro.

Le but de chaque fonction est de détecter les erreurs sémantiques et de générer le langage objet (qui, en l'occurrence, est le langage interprétatif LI).

CHAPITRE 5

REALISATION DE TESTS SUR DEUX GRAMMAIRES.

A cours de ce chapitre, nous allons montrer les tests réalisés sur les deux grammaires particulières suivantes.

5.a - Le langage externe LE/1

C'est le langage de commande utilisé par le réseau d'ordinateurs S.O.C.

Ce langage permet à l'utilisateur de spécifier ses besoins au système-réseau dans un langage général de haut niveau.

Un utilisateur peut introduire une série d'ordres constituant un "travail réseau" dans lequel il demande l'exécution d'une série d'opérations.

Ces opérations peuvent être :

1 - l'envoi d'un fichier d'un ordinateur vers un autre
2 - l'exécution d'un travail sur n'importe quel ordinateur du réseau
3 - des manipulations des fichiers (création, suppression, mise à jour, recopie,... etc.)

Le langage LE/1 (D_2) est un langage de haut niveau qui comprend :

1 - Des ordres de contrôle, destinés à établir la connexion de l'utilisateur à SOC (NETIN), et un ordre de déconnexion (NETOUT).

2 - Des instructions déclaratives, qui comprennent des déclarations de types :

- Data-set
- membre de data-set partitionné
- space
- taille de bloc physique
- d'organisation des données dans un fichier :
    - PS : séquentiel
    - DA : accès direct
    - IS : séquentiel indexé
    - PO : partitionné.

3 - Instructions exécutables, elles portent essentiellement sur des manipulations d'ensembles de données (EMPTY, DELETE, CREATE, COPY, ADD, CATALG, UNCATALG, LIST) et comprennent une commande d'exécution RUN et des opérations de calcul d'expressions arithmétiques.

Les ordres et les instructions s'écrivent à l'aide de mots-clés en tête, exception faite pour les instructions arithmétiques.

Pour donner une idée des possibilités du langage prenons des exemples de programmes en langage externe.

Un programme de ce langage s'appelle travail réseau. La première instruction d'un travail réseau doit être de la forme

NETIN liste de paramètre

où la liste de paramètres comporte le nom du travail et des renseignements de comptabilité. Le programme finit par le mot réservé NETOUT. Toutes les ressources utilisées dans le travail réseau sont déclarées par des instructions déclaratives :

CPU déclare les noms des ordinateurs concernés par le travail.

UNIT déclare les unités utilisées sur ces ordinateurs.

VOLUME déclare les volumes utilisés sur ces unités.

DSN déclare les fichiers utilisés sur ces volumes.

Exemple 1 -

```
NETIN (5161 , 0231 , 3) , (SANHOURY, TEST);
CPU  C1 := VENDOME , C2 := IMAG;
UNIT  U1 := 2314/C1; U2 := 2400/C2;
VOLUME  V1 := 002124/U1, V2 := SOCVOL/U2;
DSN  D1 := SOC . MCALIB/V1, D2 := SOC . UTIL/V2;
QUALIFY  IMAG  BY  S5161.P0003.TEST;
NETOUT
```

Exemple 2 -

Cet exemple montre le transfert d'un fichier sur disque de IMAG.67 à SAC.91.

```
NETIN (5161 , 0231 , 3) , (SANHOURY , PROG);
CPU  IMAG,67/2314/IMAG 80/A := S5161.P0003.SANH.IN;
CPU  SAC.91/2314/SAC005/B := SAC5001.SANH(OUT);
DSN  DATASET/(MBR1);
DSCB  DB := DATASET;
CREATE  B(DB);
COPY  A  TO  B;
NETOUT
```

Pour le langage externe, il comporte :

- 4è règles avec :
  - 60 symboles terminaux
  - 47 symboles non-terminaux

et l'encombrement mémoire de la grammaire, en utilisant notre méthode d'enregistrement, est de 1202 octets en comptant également les appels aux fonctions sémantiques.

5.b - Le langage PASCAL-S.

PASCAL-S (W3) est un sous-ensemble du langage de programmation PASCAL (W2), qui est destiné à l'enseignement de la programmation, pour la clarté de sa définition.

Les instructions proposées par le langage PASCAL-S se fondent sur

les expériences d'ALGOL 60.

Pascal-S contient la plupart des instructions structurées de PASCAL (instruction composée, conditionnelle, sélection et répétition).

Le langage PASCAL-S ne contient que deux types de data structurés qui sont :

1 - une structure tableau (ARRAY)

2 - une structure enregistrement (RECORD).

Les données sont représentées par les valeurs des variables. A chaque variable sont associés un identificateur et un type. Ce type peut être (entier, réel, Booléen (vraie valeur), char).

Ce type char représente le nombre de caractères à imprimer.

Un programme écrit en PASCAL-S est accepté par le compilateur de PASCAL.

Dans ce qui suit, quelques exemples représentent ce langage.

Exemple 1 -

Prenons cet exemple pour avoir une idée sur le langage PASCAL.

```
Program   test  (output);
    const ten = 10; plus = '+';
    type  row = array [1..ten] of  real;
          complex = record  re,im : real  end;
```

```
var  i,j : integer;
     p : boolean;
     z : complex;
     matrix : array [- 3..+ 3] of  row;
     pattern : array [ 1..5,1..5 ] of  char;
Procedure  dummy (var  i : integer ; var z : complex);
   var  u,v : row;
     h1,h2 : record c : complex;  r : row
   end;
function  null (x,y : real ; z : complex) : boolean;
   var   a : array  ['a'..'z']  of  complex;
       u : char;
  begin  while  x < y  do  x := x + 1.0; null := x = y
   end (* null *);
  begin  p := null (h1.c.re,h2.c.im,z)
   end  (* dummy *);
  begin  i := 85 ; j := 51 ;
   repeat
       if   i > j   then  i := i-j  else  j := j-1
        until  i = j ;  writeln (i)
   end.
```

Exemple 2 -

```
Programme  test (output);
     var  i : integer ; b : boolean;
          x : real;
     fonction  f(m,n : integer) : integer;
        begin
            f := f(n,m mod n)
        end;
        begin
              x := 9.5 ; b := true;
              i := f(511 , 31)
        end.
```

Le langage PASCAL-S comporte :

.- 45 règles

.- 48 symboles terminaux

.- 45 symboles non-terminaux.

Et l'enregistrement de la grammaire en mémoire avec notre méthode d'enregistrement occupe 676 octets y compris les appels aux fonctions sémantiques.

Dans l'annexe, il y a la représentation syntaxique de la grammaire.

5.c - Détection d'erreurs

La détection d'erreurs se fait à deux niveaux :

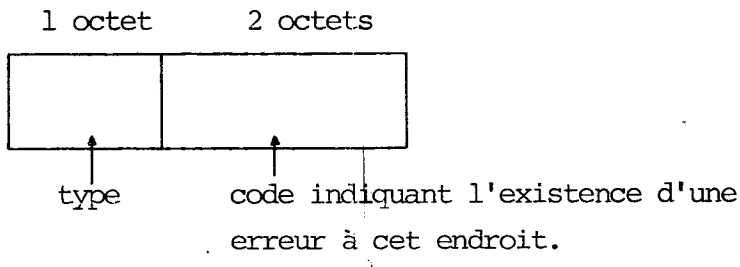
1 - niveau lexicographique.

2 - niveau syntaxique.

1 - Le premier niveau se fait pendant l'analyse lexicographique du programme donné. L'éditeur détecte les erreurs d'écriture comme la longueur d'un identificateur ou d'une chaîne de caractères, et le type de délimiteurs et de caractères spéciaux réservés par le langage.

Dans le cas de découverte d'une erreur l'analyse ne s'arrête pas, mais elle produit une unité syntaxique spéciale qui indique le type de symbole où il y a l'erreur, et la valeur de cette unité contient un code spécial qui signale à l'analyseur syntaxique qu'il y a une erreur à cette unité, pour éviter une recherche inutile.

Une unité syntaxique a la forme :



Après la construction de l'unité syntaxique, l'éditeur s'adresse à la procédure d'erreur pour sortir le message correspondant au type d'erreur et sa position dans l'instruction.

Exemple -

Prenons l'instruction suivante avec une erreur lexicographique (délimiteur faux).

```
CPU   A := IMAG.67/2314/SAC003 : B := SAC01;
                               ↑
                    symbole non correct,
```

(n'existe pas parmi les symboles réservés par le langage).

2 - Dans le cas de l'analyse syntaxique, l'analyseur essaie de détecter la plupart des erreurs syntaxiques dans les instructions d'un programme.

Dans le cas de découverte d'une erreur, l'analyseur s'adresse aux procédures d'erreurs qui sortent le message correspondant au type d'erreur et sa position dans l'instruction. Le type d'erreur est indiqué par un numéro qui, dans le cas d'erreurs syntaxiques, est fonction du code du symbole qu'il fallait trouver.

Après la détection d'une erreur, l'analyse continue pour détecter le maximum d'erreurs existant dans le programme.

Exemple -

Considérons les instructions suivantes :

```
CPU   C/2314/SAC009/INC := FAB;

SPACE  SPFAB , S1 := DSA , S2 := DSB.
                                  ↑
                                erreur
```

(une instruction doit être terminée par (;)).

Dans l'annexe on trouvera des exemples montrant l'analyse d'un programme.

CHAPITRE 6

CONCLUSION.

Le problème de la portabilité de logiciels est important dans un environnement réseau d'ordinateurs.

Dans le cadre du projet CYCLADES, réseau d'ordinateurs hétérogène reliant des machines de différentes marques, ce problème est très important pour l'implémentation de systèmes.

Le but de notre travail était de construire un système transportable flexible à coût minimum, pouvant aider à la réalisation d'un compilateur pour le langage de commande réseau qui sera utilisé par le réseau CYCLADES.

Grâce au travail que nous avons présenté ici, nous sommes arrivés, d'une part,à gagner de la mémoire en représentant la grammaire avec la méthode décrite au chapitre 3, et d'autre part, à la possibilité d'appliquer un algorithme d'analyse plus rapide.

Pour la phase d'analyse de la grammaire, nous avons essayé de détecter la plupart des erreurs syntaxiques dans un programme donné.

Nous avons également fourni un outil transportable pour aider à la définition du langage de commande au chapitre 2.

De plus, ce travail peut accepter des modifications car il est fait de procédures séparées.

Avec notre travail, qui a suivi l'approche de la transportabilité, nous espérons offrir des possibilités convenables pour les langages de commande et spécialement pour le langage de commande du réseau CYCLADES qui est encore au stade de la définition.

ANNEXE A

REPRESENTATION DE LA GRAMMAIRE DE PASCAL-S.

Notes

1. Round boxes denote symbols of the language Pascal, rectangular boxes denote syntactic constructs represented by diagrams.

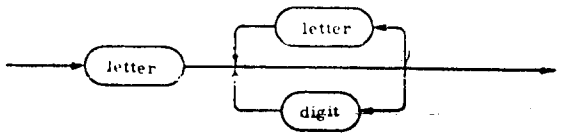
2. Separators may be inserted between any two symbols. However, no separators must occur within numbers and identifiers.

3. At least one separator must occur between consecutive identifiers, numbers, and word-symbols (such as BEGIN,END).

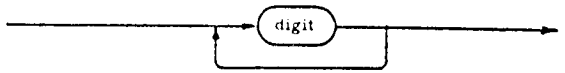
4. Separators are blanks, ends of lines, and comments. A comment is an arbitrary sequence of characters enclosed within a pair of comment brackets (* and *) .

5. The occurrence of the non-qualified word identifier in a syntax diagram implies that at this point an arbitrary, new identifier may be chosen. This identifier thereby becomes a constant-, type-, variable-, field-, function-, or procedure identifier.

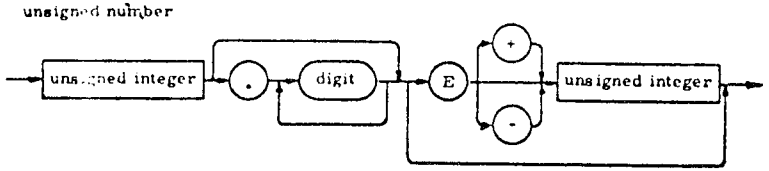
identifier



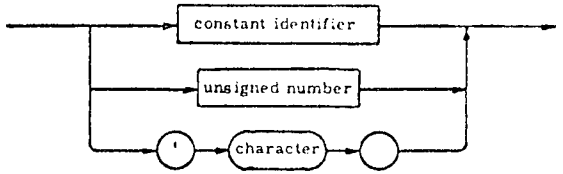
unsigned integer



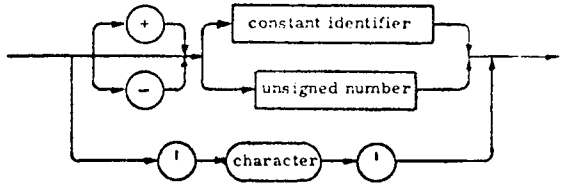
unsigned number



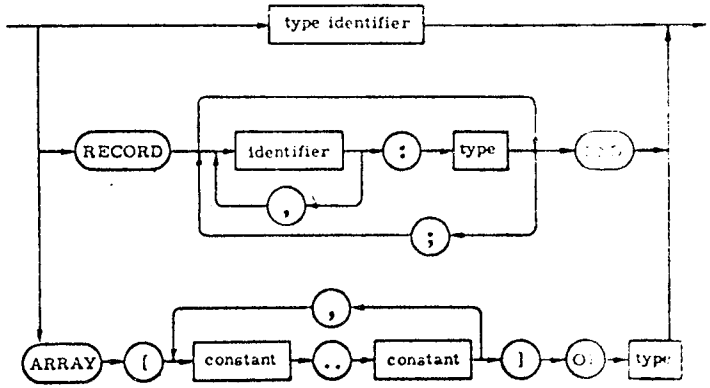
unsigned constant

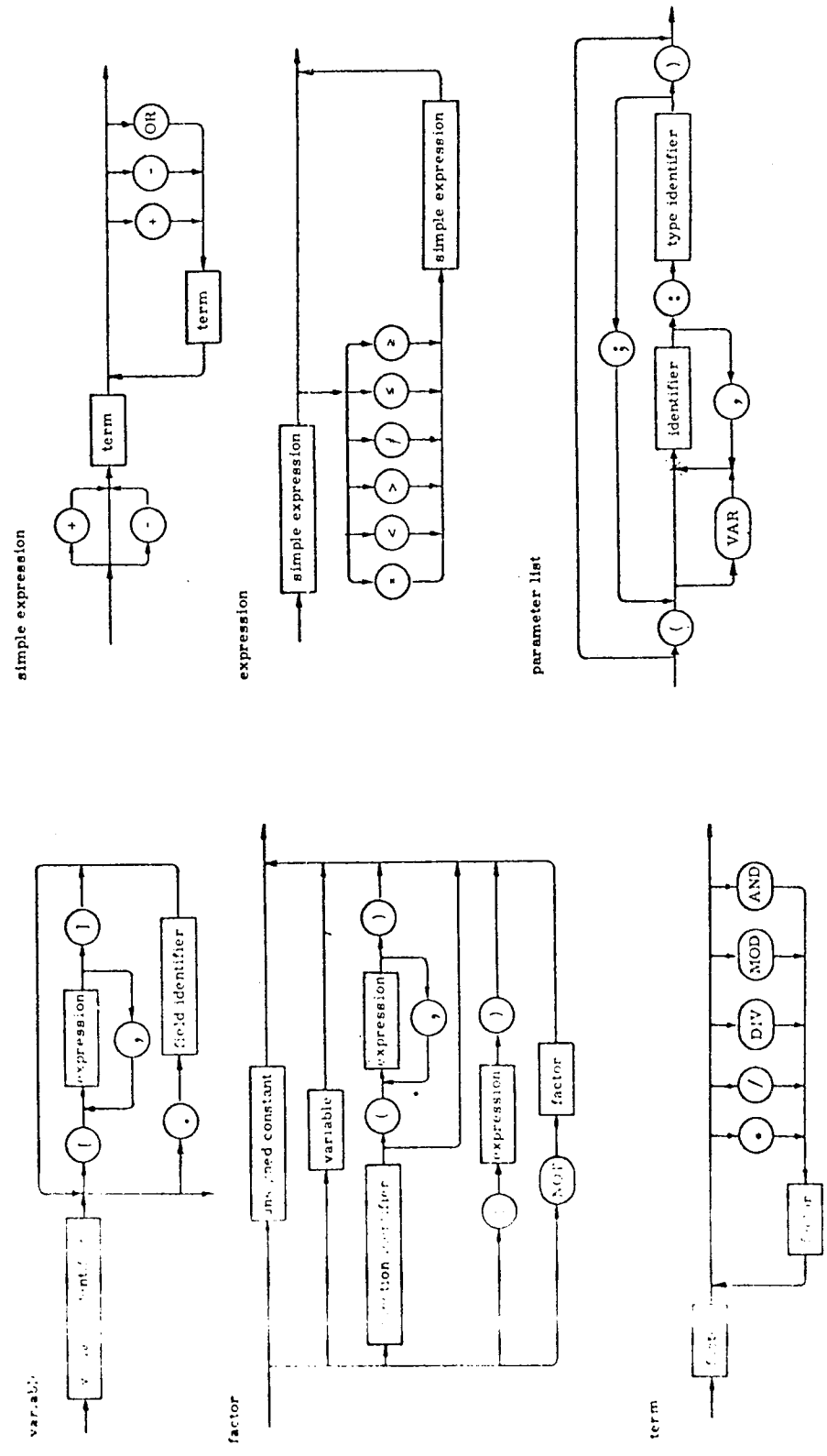


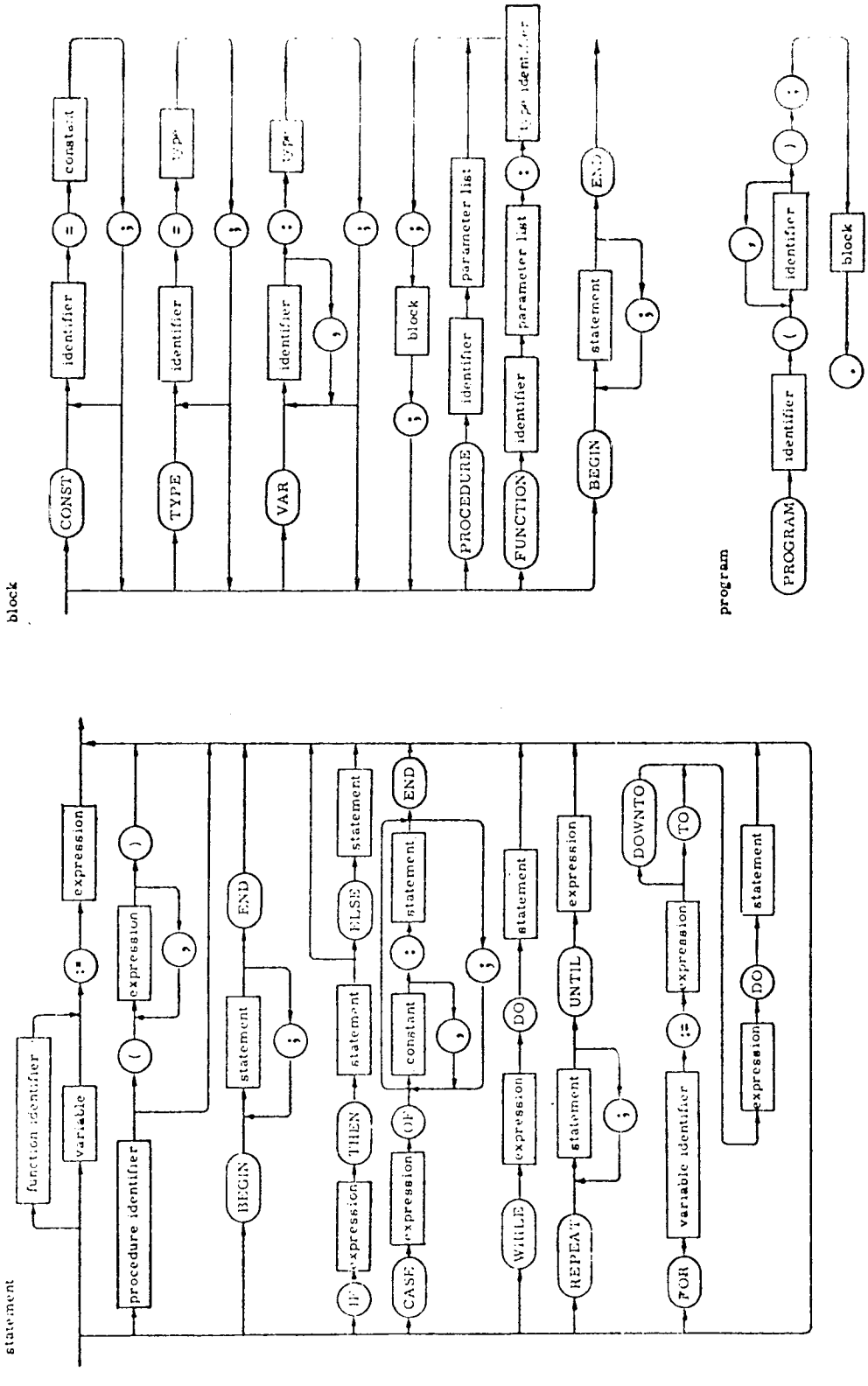
constant



type




simple expression
term
OR
+
-
term
+
-
expression
simple expression
simple expression
=
#
<
>
≤
≥
parameter list
(
identifier
,
VAR
:
type identifier
;
)
factor
(
expression
,
)
unsigned constant
variable
(
expression
)
NOT
factor
term
*
/
DIV
MOD
AND


statement
function identifier
variable
:=
expression
procedure identifier
(
expression
,
)
BEGIN
statement
;
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
,
:
statement
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
variable identifier
:=
expression
TO
DOWNTO
expression
DO
statement
block
CONST
identifier
=
constant
;
TYPE
identifier
=
type
;
VAR
identifier
,
:
type
;
block
;
PROCEDURE
identifier
parameter list
;
FUNCTION
identifier
parameter list
:
type identifier
;
BEGIN
statement
;
END
program
PROGRAM
identifier
(
identifier
,
)
;
block
.

ANNEXE B

EXEMPLES DE DETECTION D'ERREURS SYNTAXIQUES.

```
0000000011111111112222222223333333334444444445555555556666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890

NETIN (5161,0231,3),(SANHOURY,TEST) ;
CPU  A/2314/MVT004/RESA := RESULTA ;
CPU  B/2314/CIR005/RESB := RESEAU.RESULTB ;
SPACE  SPRES := (CYL,(20,10,2)) ;
BLKSIZE  B1:= 800 ;
LRECL R1:= 80 ;
 DSORG  D1 :=PS ;
REC  RA:= (FB,R1,B1) ;
DSCB  SPECIF :=  SPRES ,S1:=R1,S2:=D1 ;
CREATE  RESA (SPECIF ) ;
CREATE  RESB (SPECIF ) ;
COPY  RESA  TO  RESB ;
DELETE  RESA  ;
NETOUT
---NO ERRORS---
JOB STEP 01 TERMINATED AT  15*34*16* AFTER 0000.00 MIN
        CORE USED  0010     DISC USED  0000   WAIT TIME 0000.04
*****************************************************************************
     TIME        TIME*CORE      CORE-USE       TIME*DISC      DISC-USE
     00.00         00.25            15%          00.25           00%

      DS          DS             SHR          SHR          CR          CP          LP
   I/O-BYTES I/O-CALLS      I/O-BYTES I/O-CALLS     CARDS       CARDS       PAGES
      11264         11           9644          05         22          00          01
K 1            D5
               01
               01
            00.00
               00
               00
JOB TERMINATED SANHOURY     0001   15*34*16*
```

```
0000000001111111111222222222233333333334444444444555555555560000666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789
NETIN (5161.0231,3).(SANHOURY,TEST) ;
CPU  A/2314/MVT004/RESA := RESULTA ;
CPU  B/2314/CIR005/RESB := RESEAU.RESULTB ;
SPACE  SPRES := (CYL,(20,10,2)) ;
BLKSIZE  B1:= 800 ;
LRECL R1:= 80 ;
 DSORG  D1 :=PS ;
REC  RA:= (FB,R1,B1) ;
DSCB  SPECIF :=  SPRES ,S1:=R1,S2 ,D1 ;
*-*-ERROR 38  AT COLUMN 35
CREATE  RESA (SPECIF ) ;
CREATE  RESB (SPECIF ) ;
COPY  RESA  TO  RESB ;
DELETE  RESA  ;
NETOUT
JOB STEP 01 TERMINATED AT  17*38*05* AFTER 0000.00 MIN
.       CORE USED  0010      DISC USED  0000    WAIT TIME 0000.05
*****************************************************************************
     TIME        TIME*CORE      CORE-USE       TIME*DISC      DISC-USE
     00.00          00.24            10%          00.24            00%

      DS          DS             SHR        SHR          CR         CP         LP
   I/O-BYTES  I/O-CALLS      I/O-BYTES  I/O-CALLS     CARDS      CARDS      PAGES
       11264         11           9644         05        22         00         01
K 1             05
                01
                01
             00.00
                00
                00
JOB TERMINATED SANHOURY      0001    17*38*00*
```

```
00000000011111111112222222222333333333344444444445555555555566666666667777777773
12345678901234567890123456789012345678901234567890123456789012345678901234567891

NETIM (5161,0231,3),(SANHOURY,TEST) ;
*-*-ERROR 14  AT COLUMN  1
CPU  A/2314/MVTC04/RESA := RESULTA ;
CPU  B/2314/CIRC05/RESB := RESEAU.RESULTB ;
SPACE  SPRES := (CYL,(20,10,2)) ;
BLKSIZE  B1:= 800 ;
LRECL R1:= 80 ;
 DSORG  D1 :=PS ;
REC  RA:= (FB,R1,B1) ;
DSCB 'SPECIF :=  SPRES ,S1:=R1,S2 ,D1 ;
*-*-ERROR 38  AT COLUMN 35
CREATE  RESA (SPECIF ) ;
CREATE  RESB (SPECIF ) ;
COPY  RESA  TO  RESB ;
DELETE  RESA  ;
NETOUT
JOB STEP 01 TERMINATED AT  11*11*17* AFTER 0000.00 MIN
.       CORE USED  0010     DISC USED  0000   WAIT TIME 0000.02
**********************************************************************
     TIME        TIME*CORE     CORE-USE      TIME*DISC     DISC-USE
     00.00          00.24           15%        00.24           00%

      DS          DS            SHR         SHR          CR         CP         LP
   I/O-BYTES I/O-CALLS      I/O-BYTES I/O-CALLS     CARDS      CARDS      PAGES
       12288         12           9708         05         22         00         01
K 1            D5
               C1
               01
            00.00
               0
               0
JOB TERMINATED SANHOURY     0001   11*11*17*
```

```
0000000001111111111222222222233333333334444444444555555555566666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789

NETIN (5161,0231,3),(SANHOURY,TEST) ;
CPU  A:= IMAG.67,B:=CRICE.75,C:=SAC.91 ; ;
CPU  A/2314/V1:=MVT20 ;
QUALIFY  C  BY SAC5001.NAM ;
MOUNT  2314 ;
COPY  .A  TO  B  ;
NETOUT
---NO ERRORS---
JOB STEP 01 TERMINATED AT  11*11*17* AFTER 0000.00 MIN
.      CORE USED  0010     DISC USED  0000   WAIT TIME 0000.01
*****************************************************************************
     TIME        TIME*CORE     CORE-USE      TIME*DISC     DISC-USE
     00.00         00.17           15%         00.17          00%

      DS          DS            SHR          SHR          CR          CP          LP
   I/O-BYTES I/O-CALLS      I/O-BYTES I/O-CALLS     CARDS       CARDS       PAGES
      10240         10         10668         06         15          00          01
K 1           05
              01
              01
           00.00
              00
              00
JOB TERMINATED SANHOURY     0001   11*11*17*
```

```
0000000001111111111222222222233333333334444444444555555555566666666667777777777 8
1234567890123456789012345678901234567890123456789012345678901234567890123456 7

NETIN (11:1,0031,3),(SANHOURY,TEST) ;
CPU  A:= IMAG.67,P:=CRICE.75,C:=SAC.91 ; ;
CPU  A/2314/V1:=MVT20 ;
QUALIFY  C  BY SAC5001.NAM ;
MOUNT  2314 ;
COPY   A  TO  P  ;
*--ERROR 1 NOT COMPLETED PROGRAME
JOB STEP 01 TERMINATED AT  17*07*09* AFTER 0000.00 MIN
.        CORE USED  0010      DISC USED  0000   WAIT TIME 0000.02
*****************************************************************************
     TIME        TIME*CORE       CORE-USE       TIME*DISC      DISC-USE
     00.00         00.17              15%        00.17             00%

       DS           DS               SHR         SHR          CR          CP         LP
   I/O-BYTES I/O-CALLS      I/O-BYTES I/O-CALLS     CARDS      CARDS      PAGES
      1264          11          13568         36          15         00         01
K 1              DS
                 01
                 01
              00.00
                 01
                 00
JOB TERMINATED SANHOURY     0001   17*07*10*
```

```
0000000001111111112222222223333333334444444445555555556666666667777777771
1234567890123456789012345678901234567890123456789012345678901234567890123456789

NETIN (5161,0231,3),(SANHOURY,TEST) ;
CPU  A:= IMAG.67,B: CRICE.75,C:=SAC.91 ; ;
***-ERROR 3 SEPERATOR NOT CORRECT
***-ERROR      AT COLUMN 19
QUALIFY  C  BY SAC5001.NAM ;
COPY - A  TO  B  ;
***-ERROR 48  AT COLUMN 11
NETOUT
JOB STEP 01 TERMINATED AT  19*45*47* AFTER 0000,00 MIN
.       CORE USED  0010      DISC USED  0000    WAIT TIME 0000,01
*************************************************************************
     TIME        TIME*CORE      CORE-USE       TIME*DISC      DISC-USE
     00.00          00.16            14%          00.16            00%

      DS          DS             SHR          SHR          CR          CP          LP
  I/O-BYTES  I/O-CALLS      I/O-BYTES  I/O-CALLS     CARDS       CARDS       PAGES
      10240          10           9708          05          14          00          01
K 1             05
                01
                01
             00.00
                00
                00
JOB TERMINATED SANHOURY      0001    19*45*47*
```

ANNEXE C

EXEMPLE D'ANALYSE D'UNE GRAMMAIRE.

```
1  <NETJOB>      ::=  NETIN ; <NETPROG> NETOUT
2  <NETPROG>     ::=  BEGIN ; <NETBLOC> END ;
3  <NETBLOC>     ::=  INTEGER <ID> <FIN>
                  |   <ID> := <TERM> <FIN>
                  |   LOAD <ID> , <ID> <FIN>
                  |   <RIEN>
4  <FIN>         ::=  ; <NETBLOC>
5  <ID>          ::=  . <LETTER>
                  |   : <NUMBER>
                  |   <RIEN>
6  <LETTER>      ::=  A
                  |   B
                  |   C
                  |   D
                  |   E
7  <NUMBER>      ::=  1
                  |   2
                  |   3
8  <TERM>        ::=  ( <CODE> <ID> )
                  |   MOD <ID>
                  |   <=
9  <CODE>        ::=  +
                  |   -
                  |   *
                  |   <
                  |   %
                  |   ?
                  |   /
```

TERMINAL	FUN	REFERENCE
NETIN	F 1	, 1
;	F 2	, 1 ; 1 , 2 , 2 , 4
NETCUT	F 3	, 1
BEGIN	F 4	, 2
END	F 5	, 2
INTEGER	F 6	, 3
:=	F 7	, 3
LOAD	F 8	, 3
,	F 9	, 3
.	F10	, 5
:	F11	, 5
A	F12	, 6
B	F13	, 6
C	F14	, 6
D	F15	, 6
E	F16	, 6
1	F17	, 7
2	F18	, 7
3	F19	, 7
(	F20	, 8
)	F21	, 8
MOD	F22	, 8
<=	F23	, 8
+	F24	, 9
-	F25	, 9
*	F26	, 9
<	F27	, 9
¢	F28	, 9
?	F29	, 9
/	F30	, 9

NUMBER OF TERMINALS=30

RULE NAME		COMMENT	REFERENCE
NETJOB	10	AXIOM	, 1
NETPROG	11		, 1 , 2
NETBLOC	11		, 2 , 3 , 4
ID	01	METATERM	, 3 , 3 , 3 , 3 , 8
FIN	11		, 3 , 3 , 3 , 4
TERM	11		, 3 , 8
RIEN	01	METATERM	, 3 , 5
TD	11		, 5 , 8
LETTER	11		, 5 , 6
NUMBER	11		, 5 , 7
CODE	11		, 8 , 9

NUMBRE OF RULES=11

METATERMINALS

RIEN
IC

NUMBRE OF METATERM = 2

SEVERITY= 0

```
 0   14  * <NETJOB>          ::=  NETIN ; <NETPROG> NETOUT

$001      TERM      011            TERMINAL  NETIN
          TERM      012            TERMINAL  ;
          RULE      012,007        BRANCH TO RULE  NETPROG
          TERM      013            TERMINAL  NETOUT
          TERMT     012            TERMINAL  ;


 0   14  * <NETPROG>         ::=  BEGIN ; <NETBLOC> END

$002      TERM      014            TERMINAL  BEGIN
          TERM      012            TERMINAL  ;
          RULE      012,007        BRANCH TO RULE  NETBLOC
          TERM      015            TERMINAL  END
          TERMT     012            TERMINAL  ;


 9    8  * <NETBLOC>         ::=  INTEGER <ID> <FIN>
11   10  *                     |  <ID> := <TERM> <FIN>
15   14  *                     |  LOAD <ID> , <ID> <FIN>
40    2  *                     |  <RIEN>

$003      ALTER     009
          TERM      016            TERMINAL  INTEGER
          TERM      002            METATERM  ID
          RULET     016,031        BRANCH TO RULE  FIN
$00301    ALTER     011
          TERM      002            METATERM  ID
          TERM      017            TERMINAL  :=
          RULE      012,078        BRANCH TO RULE  TERM
          RULET     016,019        BRANCH TO RULE  FIN
$00302    ALTER     015
          TERM      018            TERMINAL  LOAD
          TERM      002            METATERM  ID
          TERM      019            TERMINAL  ,
          TERM      002            METATERM  ID
          RULET     016,003        BRANCH TO RULE  FIN
          TERMT     001            METATERM  RIEN


 0    5  * <FIN>             ::=  ; <NETBLOC>

$004      TERM      012            TERMINAL  ,
          RULET     015,044        BRANCH TO RULE  NETBLOC


 6    5  * <ID>              ::=  . <LETTER>
 6    5  *                     |  : <NUMBER>
16    2  *                     |  <RIEN>

$005      ALTER     006
          TERM      020            TERMINAL  .
          RULET     016,010        BRANCH TO RULE  LETTER
$00501    ALTER     006
          TERM      021            TERMINAL  :
          RULET     016,026        BRANCH TO RULE  NUMBER
          TERMT     001            METATERM  RIEN
```

```
 4    3  * <LETTER>          ::=  A
 4    3  *                     |  B
 4    3  *                     |  C
 4    3  *                     |  D
23    3  *                     |  E

$006       ALTER      004
           TERMT      022          TERMINAL  A
$00601     ALTER      004
           TERMT      023          TERMINAL  B
$00602     ALTER      004
           TERMT      024          TERMINAL  C
$00603     ALTER      004
           TERMT      025          TERMINAL  D
           TERMT      026          TERMINAL  E


 4    3  * <NUMBER>          ::=  1
 4    3  *                     |  2
13    3  *                     |  3

$007       ALTER      004
           TERMT      027          TERMINAL  1
$00701     ALTER      004
           TERMT      028          TERMINAL  2
           TERMT      029          TERMINAL  3


12   11  * <TERM>            ::=  ( <CODE> <ID> )
 6    5  *                     |  MOD <ID>
23    3  *                     |  <=

$008       ALTER      012
           TERM       030          TERMINAL  (
           RULE       012,017      BRANCH TO RULE  CODE
           TERM       002          METATERM  ID
           TERMT      031          TERMINAL  )
$00801     ALTER      006
           TERM       032          TERMINAL  MOD
           RULET      015,071      BRANCH TO RULE  ID
           TERMT      033          TERMINAL  <=


 4    3  * <CODE>            ::=  +
 4    3  *                     |  -
 4    3  *                     |  *
 4    3  *                     |  <
 4    3  *                     |  &
 4    3  *                     |  ?
33    3  *                     |  /

$009       ALTER      004
           TERMT      034          TERMINAL  +
$00901     ALTER      004
           TERMT      035          TERMINAL  -
$00902     ALTER      004
           TERMT      036          TERMINAL  *
$00903     ALTER      004
```

```
         TERMT    037          TERMINAL  <
$0C904   ALTER    004
         TERMT    038          TERMINAL  =
$CC905   ALTER    004
         TERMT    039          TERMINAL  ?
         TERMT    040          TERMINAL  /

000.70 SECONDS IN EXECUTION
```

REMARQUE

Lors de la génération de ces macros, les méta-terminaux sont codés par des entiers de 1 à 10 et les terminaux par des entiers à partir de 11 ; ce codage correspond à leurs arrangements dans un tableau dont les dix premiers éléments sont réservés pour les méta-terminaux.

BIBLIOGRAPHIE

ALGOL W

A1 - CHION J., CLEEMANN E.

Le langage Algol W, initiation aux algorithmes, 1973.

A2 - SUTY D.

Le langage Algol W.

Rapport de Base, IMAG, 1972.

A3 - Introduction à la programmation en Algol W.

Traduction par une équipe de l'IMAG d'un cours de H. BAURR, Fév. 1970.

B1 - BOLLIET L.

Notation et processus de traduction des langages symboliques.

Thèse , IMAG, Juin 1967.

B2 - BRASSEUR M., COHEN J.

Algorithmes d'analyse syntaxique pour langage "context free".

Chiffres, vol. 8, n° 2 et 3, 1965.

B3 - BOUSSARD J.C.

Etude et réalisation d'un compilateur Algol 60 sur calculateur électronique de type IBM 7090/44 et 7040/44.

Thèse, IMAG, Juin, 1964.

B4 - BROOKER, MACCALUM, MORRIS, ROHL

The compiler's compiler.

Third annual review of automatic programming.

Pergamon Press, 1963.

C1 - CHUPIN J.

Commande languages and heterogeneous networks.

IFIP working conference on command language, 1975.

C2 - CREVEUIL M.

Macro-mécanisme pour le langage de commande d'un système conversationnel.

Thèse, IMAG, Juillet, 1974.

CYCLADES

C3 - ANSART J.

Système interactif dans un environnement réseau.

Thèse, Février, 1976.

C4 - DU MASLE J., ANSART J.

Réseau d'ordinateurs.

Bulletin d'information du C.I.C.G. "Le Canal",

n° 18 et 19, 1974.

C5 - POUZIN L.

Présentation du réseau CYCLADES.

Bibliothèque CYCLADES.

C6 - POUZIN L.

CIGAL, The packet switching machine of the CYCLADES computer network.

Information processing, 1974.

D1 - DANISH S.

Système de contrôle du réseau d'ordinateurs S.O.C.

Thèse, Université de Paris VII, 1974.

D2 - DE CALUWE R.

Etude du langage de commande et de contrôle pour le réseau d'ordinateurs S.O.C.

Thèse, IMAG, Septembre, 1973.

D3 - DU MASLE J.

Etude d'un compilateur Algol à l'aide d'un système autocodeur.

Congrès AFIRO, Avril, 1964.

D4 - DU MASLE J.

An evaluation of the S.O.C. network commande language.

IFIP working conference on commande language, 1975.

D5 - DU MASLE J., GOYER P.

Some basic notions for computer network commande languages.

IFIP working conference on commande language, 1975.

G1 - GRIFFITHS M.

Analyse déterministe et compilateurs.

Thèse, IMAG, Octobre, 1969.

G2 - GRIFFITHS M.

Analyse syntaxique pour la production de compilateurs.

Polycopié, IMAG., 1973.

G3 - GRIFFITHS M.

Langages algorithmiques et compilateurs.

Polycopié, IMAG, 1973.

G4 - GRIFFITHS M., PELTIER M.

Grammar transformation as an aid to compiler production.

IMAG, 1968.

F1 - FLOYD R.

Syntaxic analysis and operator precedence.

JACM, 10, 3, Juillet, 1963.

F2 - FOSTER

A syntax improving device.

Computer journal, Mai, 1968.

K1 - KNUTH D.E.

Top-down syntax analysis.

ACTA Information, 1, 1971.

K2 - KOSTER

A compiler compiler.

Mathematical Centrum, Amsterdam, 1971.

LANGAGES

L1 - GIEN M., SEGUIN J.

FANNY language de macros et transportabilité.

Manuel, IMAG.

L2 - GIEN M.

Implantation du macro-processeur STAGE 2 sur CII 10070 de

l'IMAG, 1972.

L3 - RAYMOND J.

NJCL, un langage de commande pour réseau d'ordinateurs.

IMAG, Juillet, 1973.

L4 - METASYMBOL sous SIRIS7/SIRIS8.

Manuel CII d'utilisation et opérations.

L5 - Procédures systèmes sous SIRIS7/SIRIS8.

Manuel CII d'utilisation.

L6 - ICHBIAH J., RISSEN J., HELIARD J., COUSOT P.

The system implementation language LIS.

Reference Manual, 1976.

R1 - ROSEN S.

A compiler-building system developped by Brooker and Morris.

ACM, July, 1964.

S1 - SAVARY H.

Outils de mise au point pour langages de haut niveau : association de modules et contrôle de l'exécution.

Thèse, IMAG, Septembre, 1974.

S2 - SERGEANT G., FARZA M.

Machine interprétative pour la mise en oeuvre d'un langage de commande sur le réseau CYCLADES.

Thèse, Toulouse, 1974.

V1 - VIDART J.

Extensions syntaxiques dans un texte LL (1).

Thèse, IMAG, Septembre, 1974.

W1 - WIRTH N., HOARE C.

A contribution to the development of Algol.

ACM, Juin, 1966.

W2 - WIRTH N.

The programming language PASCAL.

Revised report.

International summer school on structured programming,

Munich, 1973.

W3 - WIRTH N.

PASCAL-S. A subset and its implementation.

Institut für informatik, 1975.

Z1 - ZHIRI A.

Mécanismes de base et réalisation de fonctions pour l'utilisation interactive d'un réseau d'ordinateurs.

Thèse, IMAG, Décembre, 1973.