Extensions syntaxiques dans un contexte LL(1)

Jorge Vidart

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

par

pour obtenir le grade de

Docteur de troisième cycle

Spécialité : INFORMATIQUE

Jorge VIDART

EXTENSIONS SYNTAXIQUES DANS UN CONTEXTE LL (1)

Thèse soutenue le 28 septembre 1974 devant la Commission d'Examen :

Président : Monsieur N. GASTINEL

Rapporteur : Monsieur M. GRIFFITHS

Examinateurs : Monsieur M. BERTHAUD
Monsieur A.J. COLE

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

Président : Monsieur Michel SOUTIF
Vice-Président : Monsieur Gabriel CAU

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BENOIT Jean	Radioélectricité
	BERNARD Alain	Mathématiques Pures
	BESSON Jean	Electrochimie
	BEZES Henri	Chirurgie générale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BONNIER Etienne	Electrochimie Electrométallurgie
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	CABANAC Jean	Pathologie chirurgicale
	CABANEL Jean	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Oto-Rhino-Laryngologie
	CHATEAU Robert	Thérapeutique
	CHENE Marcel	Chimie papetière
	COEUR André	Pharmacie chimique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique

Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	FELICI Noël	Electrostatique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse numérique
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GIRAUD Pierre	Géologie
	KLEIN Joseph	Mathématiques Pures
Mme	KOFLER Lucie	Botanique et Physilogie végétale
MM.	KOSZUL Jean-Louis	Mathématiques Pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques Pures
MM.	MALGRANGE Bernard	Mathématiques Pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MASSEPORT Jean	Géographie
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAUTHENET René	Electrotechnique
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET René	Servomécanismes
	PILLET Emile	Physique industrielle
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REULOS René	Physique industrielle
	RINALDI Renaud	Physique
	ROGET Jean	Clinique de pédiatrie et de puériculture
	SANTON Lucien	Mécanique
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SILBERT Robert	Mécanique des fluides
	SOUTIF Michel	Physique générale

MM.	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLAND François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
M.	VERAIN André	Physique
Mme	VEYRET Germaine	Géographie
MM.	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	BULLEMER Bernhard	Physique
	HANO JUN-ICHI	Mathématiques Pures
	STEPHENS Michaël	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

MM.	BEAUDOING André	Pédiatrie
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BONNETAIN Lucien	Chimie minérale
Mme	BONNIER Jane	Chimie générale
MM.	CARLIER Georges	Biologie végétale
	COHEN Joseph	Electrotechnique
	COUMES André	Radioélectricité
	DEPASSEL Roger	Mécanique des fluides
	DEPORTES Charles	Chimie minérale
	GAUTHIER Yves	Sciences biologiques
	GAVEND Michel	Pharmacologie
	GERMAIN Jean-Pierre	Mécanique
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	HACQUES Gérard	Calcul numérique
	JANIN Bernard	Géographie
Mme	KAHANE Josette	Physique
MM.	MULLER Jean-Michel	Thérapeutique
	PERRIAUX Jean-Jacques	Géologie et Minéralogie
	POULOUJADOFF Michel	Electrotechnique
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	ROBERT André	Chimie papetière
	DE ROUGEMONT Jacques	Neurochirurgie
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIBILLE Robert	Construction mécanique
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale

Je tiens à exprimer ma reconnaissance à:

Monsieur le Professeur Noël GASTINEL, Directeur du Centre Interuniversitaire de Calcul de Grenoble, qui m'a fait l'honneur de présider le jury de cette thèse.

Monsieur Michaël GRIFFITHS, Maître de Conférences à l'Université Scientifique et Médicale de Grenoble, qui a bien voulu diriger mes travaux, et qui par ses critiques constructives m'a permis de les mener à bien.

Monsieur Michel BERTHAUD, du Centre Scientifique IBM de Grenoble, qui a suivi mes travaux avec beaucoup d'intérêt, me faisant profiter par ses conseils de sa grande expérience.

Monsieur A. J. COLE, Directeur du "Department of Computational Science" de l'Université de St. Andrews, Ecosse, avec qui j'ai eu des discussions toujours intéressantes, et qui a accepté de juger mon travail.

Je tiens à remercier également mes amis et collègues Messieurs Paul JACQUET et Amine ZHIRI pour leurs remarques, leurs suggestions et l'analyse syntaxique, et parfois sémantique, du manuscrit de cette thèse.

Je voudrais finalement remercier le service de tirage du C.I.C.G., qui a assuré l'impression de cette thèse.

EXTENSIONS SYNTAXIQUES

DANS UN CONTEXTE LL(1)

TABLE DE MATIERES.

I N T R O D U C T I O N .

La notion de langages extensibles a été introduite en vue d'apporter une solution à la prolifération des langages de programmation pour les divers champs d'application de l'informatique. Par opposition aux langages dits universels qui essayent de fournir les moyens nécessaires pour décrire les types d'algorithmes les plus divers, les langages extensibles limitent le nombre de primitives de base et contiennent des outils pour construire les structures adéquates à chaque besoin.

Après un développement initial vers les années 70, l'étude des langages extensibles est entrée dans une période de maturation et de réflexion sur les propositions d'origine. Cette période est nécessaire afin de vérifier la validité des idées originelles et de l'outil qui était mis à la disposition de l'utilisateur; en particulier, cet outil doit être un moyen souple, sûr et efficace de construire différents dialectes.

A l'origine, les termes d'extension syntaxique et d'extension sémantique ont été introduits de manière un peu arbitraire dans la mesure où leur signification précise n'a pas été donnée.

En parlant d'extensions syntaxiques on désigne le fait de pouvoir modifier dynamiquement la grammaire d'un

langage donné afin de spécifier la syntaxe d'un nouveau langage appelé langage étendu.

Les extensions sémantiques sont celles que permettent de définir de nouveaux objets à manipuler dans le langage étendu, ainsi que les fonctions portant sur ces derniers.

Ces deux catégories d'extension ont été, en général, étudiées séparément, et ont suivi ainsi des développements différents.

Il existe plusieurs langages de programmation qui admettent des extensions sémantiques, et parmi eux le plus connu est ALGOL68; à l'heure actuelle, il existe des compilateurs pour ce langage.
Par contre on ne connaît que très peu de langages admettant des extensions syntaxiques, malgré le nombre important d'études qu'on trouve dans la bibliographie. En effet, la majorité des efforts ont été appliqués au développement de l'aspect formel et externe d'un système d'extension syntaxique, en négligeant les contraintes d'implémentation, en particulier du point de vue de l'efficacité.

Le renouveau des extensions syntaxiques, surtout en vue d'applications informatiques telles que programmation structurée, vérification de programmes, etc., et la recherche d'un système qui soit en même temps efficace à l'exécution, souple à utiliser et indépendant du langage de départ, sont à l'origine de notre travail.

Les caractéristiques du système d'extension syntaxique que nous proposons sont:

- Une notation semblable à celle de Leavenworth (23) qui constitue la manière la plus logique d'exprimer les extensions. Cette notation est dérivée de la Forme Normale de Backus, qui est communément admise pour définir des grammaires, ce qui assure ainsi la facilité d'utilisation recherchée.

- Un maximun de vérifications afin d'indiquer à l'utilisateur les erreurs éventuelles au moment de la définition de l'extension. Ceci est important pour les extensions syntaxiques car au moment d'une définition on introduit par exemple des notions nouvelles. Dans les systèmes actuels, les vérifications sont faites au moment de l'utilisation ce qui rend difficile la récupération des erreurs.

- L'utilisation d'un modèle de compilation pour la définition du système et non d'un langage particulier. Ce modèle, couramment utilisé dans des implémentations à cause de sa

simplicité, assure au système l'efficacité recherchée, et le rend indépendant du langage de base. L'insertion du mécanisme d'extension a été faite de manière à ce qu'il perturbe très peu le processus normal de compilation. Ainsi un programme écrit dans le langage d'origine n'utilise aucun composant du système chargé de traiter les extensions.

Dans le premier chapitre nous allons donner une définition des langages extensibles, et nous proposons aussi une classification formelle des extensions afin de délimiter le contexte de notre travail.

La notion de "macro" apparaît en relation avec les langages d'assemblage. L'élargissement de cette notion pour les langages de haut niveau, a conduit plusieurs auteurs à proposer des mécanismes d'extension syntaxique. Dans le second chapitre nous présentons une analyse récapitulative de ces études.

La définition de notre proposition d'un système d'extension syntaxique fait l'objet du troisième chapitre où sont développées notamment les caractéristiques externes du

système, ainsi que la méthode d'évaluation des extensions qui est fondée sur un modèle de compilateur du langage de base.

Dans le chapitre IV nous présenterons un mécanisme de construction de compilateur de langage de base. Ce compilateur se comporte selon les spécifications du modèle décrit dans le chapitre précédent, et constitue une base pour la réalisation d'extensions.

Enfin, dans le dernier chapitre nous décrivons une réalisation particulière des extensions syntaxiques pour un langage donné, qui sert à vérifier la validité des définitions et des algorithmes présentés dans les chapitres précédents. Ce prototype peut servir aussi de modèle afin d'obtenir des extensions pour d'autres langages.

C H A P I T R E I .

DEFINITION ET CLASSIFICATION DES LANGAGES EXTENSIBLES.

1.1 Définition des langages extensibles.

Le concept de langage extensible a été introduit pour rassembler une série d'études indépendantes. Un effort d'unification et de généralisation de ces travaux a été fait par Schuman et Jorrand (31) : "... il est très difficile d'attribuer les idées de base dans ce domaine à une source unique; au départ le développement a été caractérisé par un exceptionnel échange d'idées.".

La définition de langages extensibles donnée par Schuman et Jorrand est la suivante:

Un langage extensible est formé de deux composants essentiels:

1) Le langage de base , qui représente le noyau de départ de définition des extensions.

2) Des mécanismes d'extension , qui fournissent les outils de construction des extensions.

Un langage étendu est donc défini par le langage de base élargi par un ensemble d'extensions.

Les extensions sont en général divisées en deux parties: extensions syntaxiques et extensions sémantiques. Cette classification met en évidence les caractéristiques nouvelles du langage étendu. Mais aucune définition formelle délimitant clairement ces deux aspects n'a été proposée. Bert (2), en faisant une analyse des éléments constitutifs des langages de programmation, propose un schéma comportant des composants utilisés pour déterminer les champs d'application des mécanismes d'extension. Nous allons utiliser plutôt, un modèle de définition des langages, et classifier les extensions selon leur apparition dans les différentes étapes de la définition.

Dans le paragraphe suivant nous allons étudier brièvement les diverses méthodes existantes pour la définition globale des langages de programmation . Nous justifierons également le choix de la méthode qui servira de base à la définition de plusieurs types d'extensions.

1.2 Méthodes de définition des langages.

Actuellement la description d'un langage est fournie par une syntaxe, et l'étude des grammaires et des automates de reconnaissance a engendré plusieurs travaux de recherche. Ainsi on dispose d'analyseurs très efficaces pour une grande variété de langages, leur utilisation comme partie intégrante d'un compilateur étant généralisée.

Mais l'étude de la sémantique n'a pas donné de résultats comparables, dans la mesure où les modèles formels proposés n'ont jamais conduit à des implémentations opérationelles.

Dijkstra (10) propose une classification des définitions sémantiques en trois catégories ou méthodes:

Méthode interprétative. La sémantique d'un langage est définie au moyen d'une machine abstraite (interpréteur). Le comportement de cet interpréteur, étant défini pour les diverses structures du langage, renseigne sur leur signification. Conceptuellement le problème n'est pas résolu car il reste à définir cette nouvelle machine. Mais pratiquement c'est une étape car cette machine doit être assez simple et sa définition doit être moins difficile que celle du problème initial. Trois exemples illustrent cette méthode: la définition de LISP (26), la définition de PL/I par le groupe de Vienne (24), et la définition d'ALGOL68 (35). Les approches utilisées dans les trois cas pour définir la machine abstraite sont respectivement: le langage LISP lui-même; une formalisation mathématique de manipulation des arbres; et enfin une langue

naturelle: l'anglais.

Méthode fonctionnelle. Cette méthode, due à Dana Scott (32), consiste à définir le comportement d'un programme, en établissant que la "sortie" de l'exécution de chaque structure du langage est une fonction de son "entrée". Dans cette approche on s'intéresse davantage à l'ensemble des valeurs possibles des expressions, qu'à la façon dont elles sont évaluées. Du moment qu'il y a en général plusieurs manières d'implémenter une fonction, il est plus satisfaisant de spécifier d'abord mathématiquement la fonction, et après considérer son implémentation et les approximations introduites.

Méthode axiomatique. La méthode consiste à obtenir la plus faible pré-condition à l'exécution d'un programme, telle qu'à la fin de son déroulement, une post-condition donnée soit satisfaite. Ainsi par exemple, pour l'instruction d'assignation:

x=x+1

si on impose les post-conditions:

p1: a<7 p2: x<10

les plus faibles pré-conditions valides avant l'exécution de l'instruction sont respectivement:

fs1: a<7 fs2: x<9

Le fait que l'affectation concerne seulement la variable "x" entraîne la variation de la deuxième condition et laisse la première inchangée. L'idée c'est que si pour chaque structure exécutable du langage, on peut trouver les pré-conditions correspondantes à une post-condition quelconque, alors on dispose de toute l'information sur le comportement de cette structure.

Les deux dernières méthodes, sont surtout utilisées pour la vérification des programmes et des analyses de comportements, mais elles n'ont pas servi à définir des langages. Il existe des applications a posteriori, comme la définition de PASCAL (17) faite par la méthode axiomatique.

L'utilisation des méthodes interprétatives est très répandue, et parmi elles, la méthode dite de Vienne est la seule qui délimite clairement les tâches d'un compilateur réel, et qui peut être utilisée comme schéma d'une implémentation. C'est pour cela, et malgré les critiques conceptuelles qu'elle appelle, que nous allons la prendre comme base.

1.2.1 Méthode de "Vienne".

La méthode de Vienne, schématisée dans la figure 1.1, peut être considérée comme une définition interprétative des langages. Elle utilise le concept de syntaxe abstraite et de machine abstraite. Le premier introduit par McCarthy (27), est utilisé dans la méthode comme point de départ de la définition, et exprime les élements du langage comme des structures adéquates à l'interprétation. Le second concept comprend un ensemble d'états et une fonction de transition qui spécifie les conditions de passage entre les états. Un programme et ses données définissent un état initial, et le comportement de la machine abstraite constitue l'évaluation de ce programme pour ces données.

Pour l'utilisation du langage on définit aussi une syntaxe concrète qui décrit l'aspect externe comme une suite de caractères d'un certain vocabulaire.

On pourrait imaginer une machine attachée directement à la syntaxe concrète; mais pour la plupart des langages de programmation, il est plus pratique de "nettoyer" un programme de ses contraintes d'écriture, afin de faciliter la tâche de l'interpréteur. Ainsi on introduit la fonction "ACCEPTER" , qui construit l'arbre syntaxique d'un programme, et qui réalise le rôle classique d'un

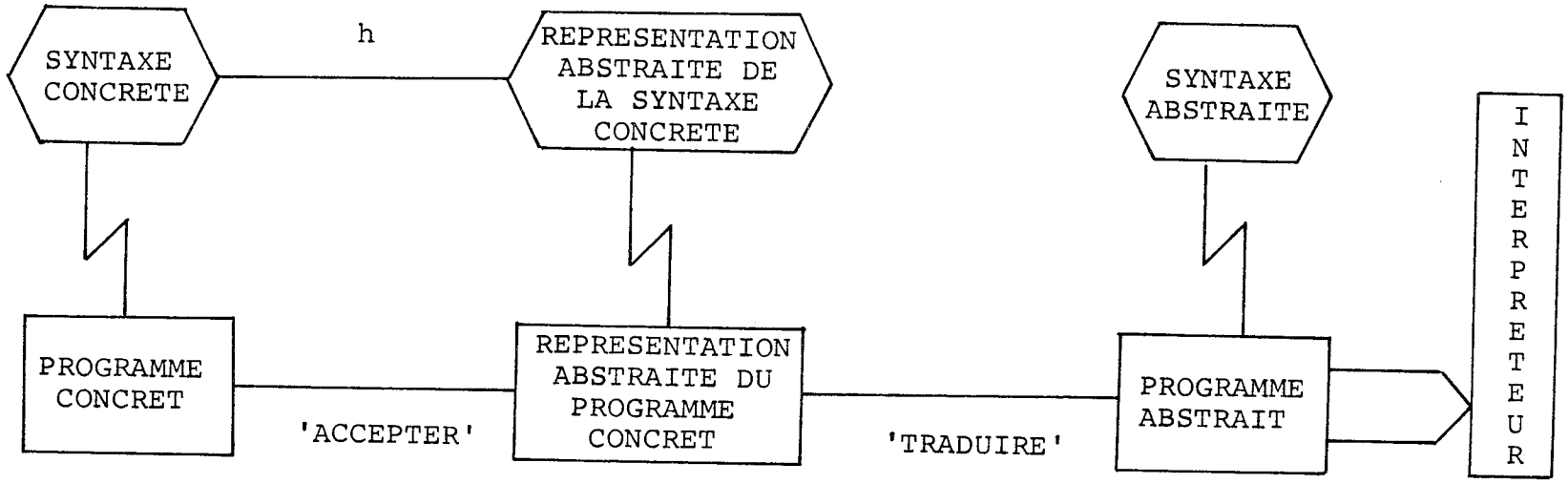


FIGURE 1.1

Schéma de la Méthode de 'Vienne'.

automate d'acceptation. Cet arbre nommé représentation abstraite du programme concret, est un objet structuré comme une arborescense à branches nommées, type de structure qui est utilisée uniformément dans toute la méthode. Cet objet doit satisfaire une suite de prédicats qui constituent la représentation abstraite de la syntaxe concrète. La fonction "h", qui apparaît dans la figure, produit automatiquement ces prédicats à partir de la syntaxe concrète.

Cette nouvelle représentation de la syntaxe pourrait servir de base pour la définition de l'interpréteur, cependant, à nouveau, des raisons pratiques et non conceptuelles, conduisent à réaliser des actions supplémentaires avant l'interprétation. Ainsi on incorpore une fonction "TRADUIRE" chargée de la vérification de la sémantique statique du langage: manipulation des déclarations, relations entre les occurrences de définition et d'application des objets, vérification des conditions de contexte, etc (Griffiths (15)).

Le résultat de la fonction "TRADUIRE" est un objet structuré selon Vienne, appelé programme abstrait.Cet objet vérifié par la syntaxe abstraite représente l'entrée de l'interpréteur.

1.3 Classification des extensions.

Nous allons maintenant donner une classification des extensions basée sur le modèle ci-dessus. Chaque type d'extension sera en relation avec une des parties composant le modèle.

1.3.1 Extensions syntaxiques.

On dira qu'une extension est syntaxique lorsque le champ d'application des mécanismes d'extension est la syntaxe concrète du langage de base.

Cette définition est en concordance avec le concept classique selon lequel les extensions syntaxiques servent à modifier l'aspect externe du langage.

L'application du modèle de définition au langage étendu donne comme différence avec celui du langage de base, une syntaxe concrète modifiée, et une fonction "ACCEPTER" élargie de façon à reconnaître les nouvelles structures syntaxiques.

1.3.2 Extensions sémantiques.

Les extensions sémantiques sont celles qui servent à modifier la syntaxe abstraite d'un langage.

L'extension classique est l'extensibilité des données, qui comporte deux phases séparées:

a) Définition des nouveaux types de données (ou classes (20), ou modes (35)).

b) Définition de nouveaux opérateurs, qui admettent comme opérandes, des élements dont les types sont soit primitifs, soit composés ou définis à partir de ces derniers.

Ces deux phases ont des répercussions différentes dans la définition du langage étendu:

a) L'introduction des nouveaux types est essentiellement statique, et implique des modifications à la fonction "TRADUIRE", afin d'effectuer les vérifications. On les appellera "extensions à la sémantique statique".

b) La définition des nouveaux opérateurs implique des actions nouvelles à accomplir lors de l'exécution d'un programme du langage étendu, donc on doit modifier l'interpréteur. On les appellera "extensions à la sémantique dynamique".

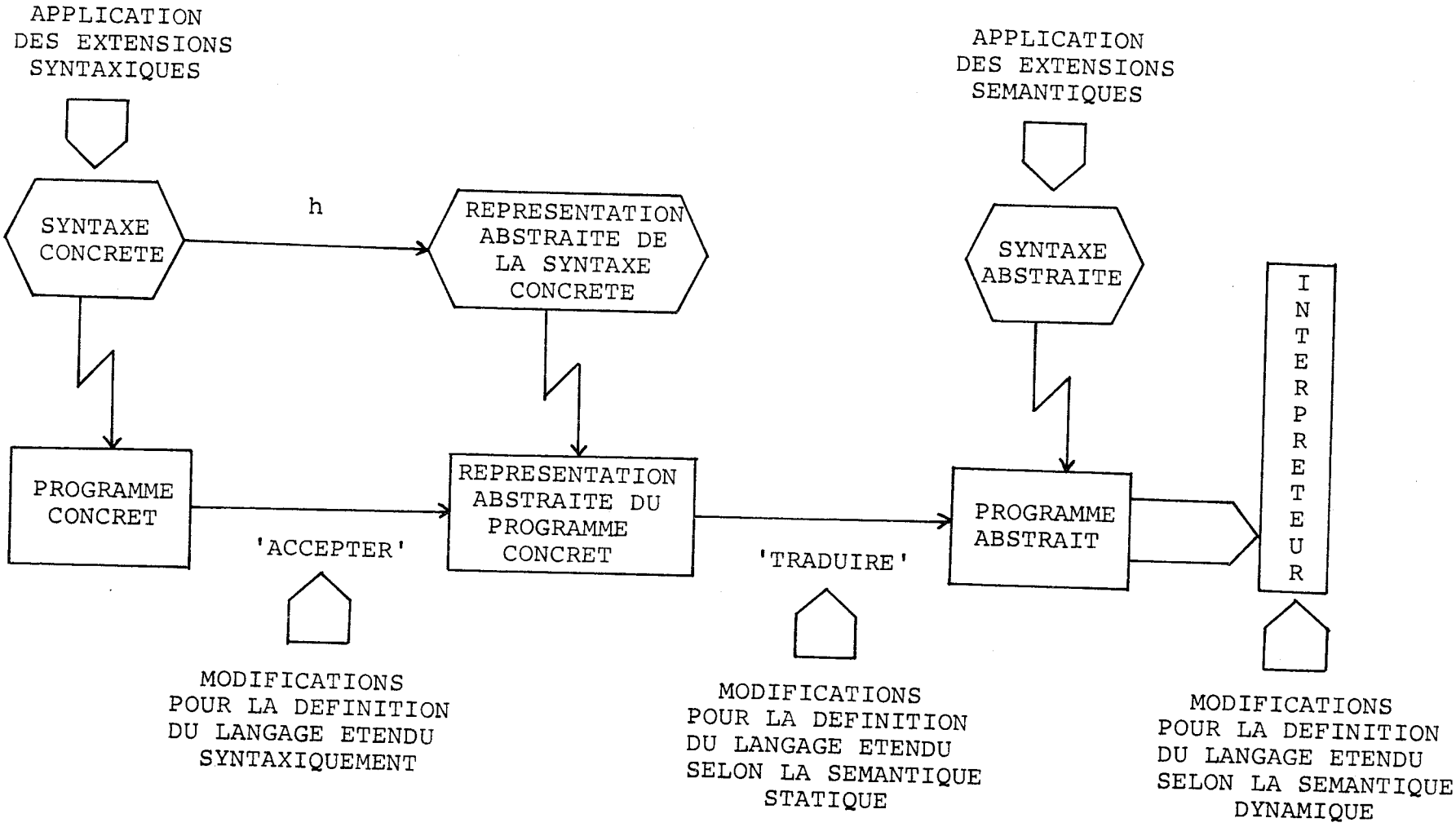


FIGURE 1.2

Schéma de définition de langage
après l'introduction des extensions.

Dans la figure 1.2 on a indiqué tous les cas possibles des modifications à introduire, afin d'obtenir les définitions des langages étendus à partir du langage de base.

Une fois les définitions précisées nous nous intéresserons exclusivement dans ce qui suit à un système d'extension syntaxique. Nous présenterons ainsi, des mécanismes permettant de modifier la syntaxe concrète d'un langage donné.

C H A P I T R E I I .

MACROS ET LANGAGES DE HAUT NIVEAU.

Nous allons analyser maintenant diverses propositions d'extension syntaxique. Nous suivrons l'évolution des études sur le sujet, à partir des macro-assembleurs jusqu'aux langages extensibles.

2.1 Macro-assembleurs.

On peut considérer que les macro-assembleurs constituent une première méthode d'extension. Un langage d'assemblage auquel on ajoute un ensemble de macros, dont l'expansion est une suite d'instructions du langage de base constitue un langage étendu.

Une définition de macro comporte un nom et un corps. Le nom identifie la macro et représente une instruction du langage étendu. L'utilisation d'une telle instruction est nommée "appel" de la macro-instruction. Le corps est constitué par une suite d'instructions du langage d'assemblage et des appels de macros définies préalablement. Avec ces caractéristiques la différence entre le langage étendu et le langage de base est essentiellement syntaxique.

Du point de vue de l'utilisateur les possibilités

offertes par les macro-assembleurs, peuvent paraître analogues à celles des sous-programmes admis par la plupart des langages d'assemblage. La différence entre les deux solutions est que les sous-programmes sont évalués lors de l'exécution du programme, tandis que l'évaluation des appels de macros est faite lors d'une étape préalable à cette exécution. Ainsi la "compilation" d'un programme du langage étendu comporte deux étapes:

- évaluation des macros, en remplaçant les appels par leurs corps respectifs.
- assemblage du programme résultant.

2.2 Macro-processeurs.

Une approche analogue aux macro-assembleurs, peut être envisagée pour les langages de haut niveau avec les macro-processeurs.

Brown (5) définit les macro-processeurs comme "des systèmes de programmation qui fournissent à l'utilisateur les moyens d'ajouter des facilités nouvelles créées par lui, à un autre système déjà existant".

A tout macro-processeur est associé un macro-language qui détermine la syntaxe des appels de

macros, des définitions, et éventuellement des instructions de contrôle. Mais le système se comporte plus comme un processeur que comme un langage de programmation courant, les textes à évaluer ne sont pas seulement composés de l'utilisation de son macro-language, mais peuvent comporter aussi des sections inconnues du processus,et donc, transparentes à l'évaluation.

McIlroy (28), est le premier qui, en rassemblant des idées d'origines diverses, utilise les macro-processors pour les langages de haut niveau. A la suite de son travail, d'autres propositions on été faites; parmi les plus connues on trouve: ML/I (6), GPM (33) (dont nous avons fait une description dans (36)), TRAC (29), LIMP (37), etc. Ce dernier introduit l'idée de syntaxe variable pour les appels ("template"). Avec ce mécanisme l'utilisateur a la liberté de définir le texte qui correspondra à un appel de macro. Ce concept a été utilisé dans d'autres macro-processeurs (MP/3 (25)), et aussi a servi de base à la proposition de Leavenworth (23) que nous analyserons par la suite.

L'application des macro-processeurs aux extensions syntaxiques consiste à définir des macros dont l'expansion appartient à un certain langage de base. Ici le problème est plus complexe qu'avec les langages d'assemblage, car la syntaxe du langage de base sera plus élaboré. Les macro processeurs sont en géneral utilisés pour l'obtention de

nouvelles instructions. Pour pouvoir étendre les autres classes syntaxiques, il faudra que le macro-processeur connaisse la syntaxe du langage de base. Ceci nous amène à distinguer les macro-processeurs généraux, applicables à n'importe quel langage, mais avec des objectifs d'extensions limités, des macros-processeurs spéciaux, conçus pour un langage de base donné.

2.3 Macros syntaxiques.

Une nouvelle approche a été faite par Cheatham (8) en définissant un système où les macros dépendent non seulement du langage de base, mais en plus de la manière dont ce langage de base sera implementé. Ainsi trois types de macros sont définis:

a) macros de texte. Ces macros sont implementées au moyen d'un pré-processeur du compilateur du langage de base.

b) macros syntaxiques. Ces macros sont évaluées lors de la phase d'analyse syntaxique du compilateur. Pour la définition des macros le programmeur utilise les classes syntaxiques du langage de base.

c) macros de génération. Ces macros sont évaluées lors de la phase de génération de code

du compilateur.

Le rôle des macros de texte est le même que celui accompli par les macro-processeurs. Afin de définir la place dans le processus de compilation, où sont traités les autres types de macros, Cheatham se sert d'un modèle de compilateur. Ce compilateur est dirigé par la syntaxe et se compose de plusieurs phases.

Dans le système de Cheatham on trouve pour la première fois l'idée d'extension sémantique à travers des facilités offertes par les macros de génération. Quant à l'extensibilité syntaxique, étant insérée dans le processus de reconnaissance du compilateur, elle peut s'appliquer à toutes les classes syntaxiques du langage de base.

Voyons au moyen d'un exemple les caractéristiques du système syntaxique.

```
LET N BE INTEGER
SMACRO MATRIX(N) AS ATTRIBUT
MEANS 'ARRAY(1:N,1:N)'
```

ATTRIBUT est une classe syntaxique du langage de base. Un appel de macro a la forme suivante:

MATRIX(argument)

Cet appel ne sera évalué que si le processus de reconnaissance du texte source est en train d'analyser la classe "ATTRIBUT" (Par la suite cette situation sera appelée

"contexte de la clase syntaxique" pendant l'analyse). Dans ce cas le texte suivant sera substitué à l'appel:

ARRAY(1:argument,1:argument)

Schuman et Jorrand (31) proposent deux modèles d'extension. L'un sémantique, rassemble les idées d'extensibilité des données qu'on trouve dans BASEL (19). L'autre, syntaxique, étend les idées de Cheatham en ajoutant aux systèmes des macros, un ensemble de prédicats pour diriger le processus de substitution. A partir de ces idées, Chatelin et Willis ont implementé un macro-processeur syntaxique (7). Ce macro-processeur travaille à partir d'une syntaxe qui décrit le langage de base sous une forme BNF. Les définitions de macros sont des règles de production qu'on ajoute à la grammaire du langage de base, suivies par le texte de remplacement. Le processus d'évaluation est semblable à celui défini par Cheatham. La puissance de ce macro-processeur se trouve dans sa souplesse à manipuler les arbres syntaxiques lors de l'évaluation des appels. La conception de ce macro-processeur a été dirigée vers une application dans un système de définition des langages (38).

Presqu'en même temps que le travail de Cheatham, d'autres propositions de langages extensibles ont été faites: Leavenworth (23), Galler et Perlis (13).

Leavenworth présente un système de macros syntaxiques applicable à des compilateurs dirigés par la

syntaxe. Ses objectifs sont plus limités que ceux de Cheatham car les seules classes syntaxiques qu'il peut étendre sont celles des "instructions" et des "primaires", existant dans la plupart des langages de programmation. Cette restriction tient au fait que Leavenworth veut fournir, au moyen des macros, des possibilités semblables à ce qu'on obtient avec l'utilisation des procédures dans les langages de haut niveau. L'importance de la méthode réside dans la manipulation des paramètres. Un paramètre aura un nom, mais aussi doit appartenir à l'ensemble défini par un type syntaxique du langage de base. La vérification d'un appel se fera de la manière classique, mais on exige, en plus, la vérification syntaxique des paramètres. Nous avons réalisé dans (36) une description de cette méthode; et ne présenterons ici qu'un exemple.

Rappelons tout d'abord que la définition d'une macro comporte deux parties:

a) La structure qui décrit la syntaxe du texte source à reconnaître.

b) Le corps de la macro.

Pour que l'analyseur reconnaisse une définition de macro on introduit trois séparateurs:

smacro pour commencer la définition.

define pour séparer la structure du corps.

endmacro pour indiquer la fin de la définition.

Les noms de classes syntaxiques représentent les paramètres formels de la macro. Ces paramètres sont dénotés

dans le corps au moyen du symbole "$", suivi d'un numéro correspondant à leur ordre d'apparition dans la structure.

Nous allons définir une instruction itérative du type "pour", dans un langage qui ne dispose comme instructions de contrôle que d'instructions conditionelles, et d'instructions de branchement:

```
smacro for <variable> := <express>
           to <express>
           do <statement>
define begin $1 := $2 ;
             L1: if $1 = $3 then
                                begin
                                 $4 ;
                                 $1:=$1+1 ;
                                 goto L1 ;
                                end
        end
endmacro
```

2.4 Caractéristiques essentielles d'un système d'extension syntaxique.

Nous essayerons de dégager les caractéristiques essentielles dont on doit tenir compte pour la formulation d'un système d'extension syntaxique.

2.4.1 Emplacement dans le compilateur

Un système d'extension syntaxique doit admettre des extensions pour toutes les classes syntaxiques du langage de base. Pour éviter une phase d'analyse syntaxique supplémentaire lors d'un pré-passage, il est souhaitable de l'incorporer dans le compilateur. La reconnaissance faite par le compilateur sert en même temps à déterminer les appels et procéder à leur évaluation.

2.4.2 Définitions des macros.

Dans toutes les propositions, la modification de la syntaxe du langage de base produite par les mécanismes d'extension consiste à ajouter une nouvelle production à une classe syntaxique déjà existante. Ainsi une définition est composée de trois parties:

- Nom de la classe syntaxique à étendre.

- Texte à reconnaître lors d'un appel, et qui joue le rôle du nom de la macro.
- Texte de substitution.

Pour le traitement des paramètres la solution la plus intéressante est celle de Leavenworth. Les paramètres ont un nom, mais doivent aussi appartenir à l'ensemble des terminaux dérivés d'une classe syntaxique. La vérification de cette appartenance fait partie du processus de validation d'un appel. Le passage des paramètres se fait par "valeur" dans toutes les propositions.

2.4.3 Evaluation des appels.

Les tentatives de vérification des appels ne vont être faites que si l'analyse syntaxique est placée dans le contexte de la classe syntaxique à étendre. Ceci est une caractéristique essentielle qui marque la différence avec les macro-processeurs généraux. Après la reconnaissance de l'appel l'évaluation se réalise d'une façon classique, en remplaçant le texte, qui représente l'appel, par le corps de la macro. Le processus d'analyse syntaxique est poursuivi avec le nouveau texte.

Dans le chapitre suivant nous définirons notre

proposition qui est basée sur les considérations qu'on vient de décrire.

Brown, dans son travail (5), regrette qu'aucun système d'extension syntaxique ne soit implémenté. Nous avons dit que Chatelin et Willis ont fait une implémentation d'un macro-processeur syntaxique, mais comme un système indépendant, et son application à un compilateur réel n'a pas été envisagée.

Cette absence d'implémentation étant due à l'inefficacité a priori des diverses propositions, nous avons été amenés à définir un système qui réalise les évaluations d'une manière différente de celles qui ont été exposées. Ainsi le concept de macro disparaît, laissant la place à la manipulation de programmes abstraits.

C H A P I T R E I I I .

UN SYSTEME D'EXTENSION SYNTAXIQUE.

Nous allons définir dans ce chapitre un système d'extension syntaxique. Dans une première partie nous présenterons ses caractéristiques externes, qui incluent les outils fournis à l'utilisateur afin de définir et d'utiliser un langage étendu. Dans une deuxième partie nous exposerons la méthode d'évaluation dont dispose le système, et nous décrirons les relations entre le compilateur du langage de base et les mécanismes d'extension. Dans une dernière partie nous allons définir les composants du système et analyser la dynamique du processus d'évaluation des extensions.

3.1 Définition et utilisation des extensions syntaxiques.

Dans un environnement d'extensibilité, un programme non seulement remplit la fonction classique de décrire un algorithme, mais en plus définit le langage où lui même est valide. On conservera le mot "programme" pour nommer les utilisations d'un langage extensible, mais il faudra tenir compte du sens spécial qui lui est attaché selon les deux tâches différentes à réaliser.

Lors de l'utilisation d'un langage extensible il y a

deux étapes à considérer:

a) Définition d'une extension. A ce niveau on étend le langage en créant de nouvelles structures syntaxiques, et en fournissant des renseignements sur leurs significations.

b) Utilisation d'une extension définie préalablement. Cela signifie qu'on programme dans le langage étendu.

3.1.1 Définition des extensions.

Nous avons dit qu'un langage étendu est défini par un ensemble d'extensions ajoutées à la définition du langage de base.

La définition du langage étendu est un processus dynamique qui est activé à chaque définition d'extension. On peut considérer que dans un programme on a défini autant de langages étendus, qu'il existe de définitions d'extensions; chacun de ces langages a comme sous-ensemble strict les langages associés aux définitions d'extension précédentes dans le programme.

Une définition d'extension syntaxique est équivalente à la création d'une nouvelle règle de production pour une classe syntaxique du langage de base. Cette règle de production, qu'on appellera structure d'une définition comporte l'aspect "syntaxique" d'une extension syntaxique.

Mais pour élaborer un programme dans le langage étendu il faut spécifier aussi la sémantique de ces structures, qu'on appellera la signification d'une définition. Cette signification représente ce qui pour les propositions que nous avons présentées dans le chapitre précédent, était le corps de la macro syntaxique.

Pour décrire la signification on a besoin d'un langage. On pourra imaginer un langage ad-hoc, créé spécialement pour cette fonction. Mais il est souhaitable d'utiliser le langage étendu valable au moment de la définition pour deux raisons. Primo, l'utilisateur n'a pas besoin d'apprendre deux langages différents et de les utiliser dans un seul et même programme. Secundo, l'introduction d'un deuxième langage implique la co-existence de deux "processeurs" différents, ce qui complique l'implémentation et risque de diminuer l'efficacité du système global.

Nous adopterons la forme suivante pour une définition d'extension:

EXTSYN classe-à-étendre := structure|->signification FINEXT

Nous conviendrons que la syntaxe du langage de base est définie par une grammaire et décrite selon la méthode BNF. La grammaire est donc définie de façon classique par un 4-uple:

- Vocabulaire non-terminal (ou ensemble des classes syntaxiques).
- Vocabulaire terminal.
- Ensemble de règles de production.
- Axiome.

Dans ce qui suit nous allons analyser les différents composants d'une définition d'extension. Nous montrerons aussi leur utilisation, en présentant comme exemple la définition des primitives de manipulation de listes pour un langage qui ne dispose pas de cette facilité.

3.1.1.1 Structure d'une définition.

Au moyen de la structure on va créer une nouvelle alternative de génération pour une classe syntaxique du langage de base. La syntaxe des langages de programmation étant définie au moyen d'une grammaire, il est souhaitable d'utiliser une notation pour décrire les structures, semblable à celle dont on dispose pour la description des grammaires. Ainsi une structure peut être imaginée comme la partie droite d'une règle de production, dont la partie gauche correspondra à la classe syntaxique à étendre. De cette façon la structure sera représentée comme une suite d'éléments terminaux et non terminaux du langage de base.

Pour des raisons que nous verrons lors de l'analyse de la méthode d'évaluation le premier composant de la structure doit être un élément terminal, qui devient ainsi un mot-clé de l'extension. Ceci est en rapport avec l'algorithme d'analyse syntaxique que nous utiliserons dans le compilateur.

Les classes syntaxiques qui apparaissent dans la structure dénotent les paramètres formels de l'extension. Pour qu'une utilisation d'extension soit correcte les paramètres réels doivent vérifier les types syntaxiques des paramètres formels correspondants.

3.1.1.2 Signification d'une définition.

La signification représente la sémantique de l'entité syntaxique définie par la structure. D'après ce que nous avons déjà dit le langage d'expression de la signification sera le langage étendu, valable à l'endroit de la définition. Mais on va imposer des contraintes syntaxiques à la signification: elle doit être une dérivation correcte de la classe syntaxique objet de l'extension. Ceci est en liaison avec le processus d'évaluation. Au début de la reconnaissance d'une utilisation l'analyse syntaxique faite par le compilateur se trouve dans un certain contexte qui doit être respecté

après l'évaluation. Ainsi, dans l'exemple:

IF NULL(x) THEN .....

NULL(x) étant une utilisation d'extension, après son évaluation le compilateur doit continuer l'analyse de l'expression conditionnelle commencée avec "IF".

Structurellement la signification est aussi composée d'une suite d'éléments terminaux et non-terminaux. Ces derniers représentent les paramètres formels de l'extension et ils sont en concordance avec ceux de la structure.

3.1.1.3 Manipulation de listes.

Nous allons présenter maintenant des exemples de définitions d'extensions syntaxiques. Notre objectif est de définir les primitives de manipulations de listes pour le langage GSL. GSL(4) est un langage d'écriture de systèmes qui offre certains avantages des langages de haut niveau, et les possibilités d'accès aux ressources de la machine. Les caractéristiques significatives du langage qu'il faut rappeler ici pour une meilleure compréhension des exemples sont les suivantes:

- On peut déclarer des variables de type "pointeur", dont les valeurs sont des adresses en mémoire de donnees du programme.
- Une variable peut être déclarée comme ayant

la classe de mémoire "BASED". Dans ce cas l'allocation de mémoire de ces variables est sous le contrôle de l'utilisateur.

- On dispose de l'opérateur de qualification "->" qui sert à associer une adresse mémoire à une variable BASED.
- On peut organiser les données dans des structures analogues à celles de PL/I.

Les fonctions que nous allons définir représentent les primitives du langage LISP (26). Chaque composant d'une liste est un doublet de valeurs. Nous utilisons une zone spéciale pour ranger les éléments des listes.

```
    /+ On définit une structure pour travailler avec les
doublets +/
        DECLARE 1 CELLULE BASED,
                  2 TETE POINTER,
                  2 QUEUE POINTER ;
    /+      ++++++++++++++++++++++++++++++++++++      +/

/+ Fonctions permettant l'accès aux éléments d'une liste +/
EXTSYN <elem> := CAR(<elem>.1) |-> (<elem>.1->TETE) FINEXT;
EXTSYN <elem> := CDR(<elem>.1) |-> (<elem>.1->QUEUE) FINEXT;
    /+      ++++++++++++++++++++++++++++++++++++      +/
```

```
      /+ Predicat pour détecter la fin d'une liste +/
EXTSYN <exprif> := NULL<elem>.1|-><elem>.1=ADDR(NIL) FINEXT;
    /+      ++++++++++++++++++++++++++++++++++      +/

/+ Les valeurs atomiques seront les données courantes du
programme. Donc en analysant un élément d'une liste il faut
pouvoir tester, s'il "pointe" vers une liste ou vers un
atome +/

          /+ Fonction d'adressage des atomes +/
EXTSYN <elem>:= ADRATOM(<ident>.1)|->
                    ADDR(<ident>.1)|'FF000000'X FINEXT ;
    /+      ++++++++++++++++++++++++++++++++++      +/

     /+ Predicat pour tester les éléments atomiques +/
EXTSYN<exprif>:= ATOM(<elem>.1)|->
         (<elem>.1 & 'FF000000'X )= 'FF000000'X FINEXT ;
     /+      ++++++++++++++++++++++++++++++++++     +/

     /+ Fonction pour accéder à la valeur d'un atome +/
EXTSYN <elem>:= VALATOM(<elem>.1) |->
           ((<elem>.1 |'FF000000'X) ->ATOMIQUE) FINEXT ;
     /+      ++++++++++++++++++++++++++++++++++     +/
```

```
        /+ Fonction pour la composition des listes +/
EXTSYN <stlist> := CONS(<elem>.1,<elem>.2) DANS <elem>.3 |->
           <elem>.3=LIBRE;
           LIBRE=CDR(LIBRE);
           IF ATOM(<elem>.1)
               THEN ADRATOM(<elem>.1)=>CAR(<elem>.3)
               ELSE <elem>.1=>CAR(<elem>.3);
           IF ATOM(<elem>.2)
               THEN ADRATOM(<elem>.2)=>CDR(<elem>.3)
               ELSE <elem>.2=>CDR(<elem>.3);
FINEXT;
     /+       ++++++++++++++++++++++++++++++++++++      +/
```

On désigne par "elem" la classe syntaxique qui dans la plupart des langages de programmation apparaît lors de la génération des expressions au niveau le plus bas. Ceci veut dire qu'elle peut figurer comme opérande de tous les opérateurs primitifs, et pour les langages dérivés d'ALGOL60 elle représente une constante, ou une variable ou une expression parenthésée. La classe "exprif" représente les prédicats qu'on peut définir dans le langage de base. La classe "stlist" génère une suite d'instructions.

On utilise l'adresse de la variable NIL pour indiquer la fin d'une liste. La variable ATOMIQUE a la classe de mémoire BASED et elle sert a accéder à un atome. LIBRE est une liste qui rassemble les doublets libres . Lors

de l'acquisition d'un doublet, tel qu'elle a été faite dans l'extension CONS, on met à jour la liste LIBRE. Avant la réalisation de cette mise à jour, il faut vérifier que l'on dispose encore de doublets libres. Si ce n'est pas le cas, on doit exécuter un algorithme de "ramassage de miettes" que nous n'avons pas inclus dans le programme pour des raisons de clarté.

3.1.2 Utilisation des extensions.

Utiliser une extension consiste à programmer dans le langage étendu. Donc on doit respecter les règles établies lors de la définition, d'une manière analogue à ce qu'on doit faire pour n'importe quel langage. Pour qu'une utilisation soit correcte il faut:

- Insérer l'utilisation dans le contexte de la classe syntaxique objet de l'extension.
- Respecter la syntaxe décrite par la structure.
- Fournir les paramètres effectifs nécessaires en accord avec le type syntaxique des paramètres formels qui apparaissent dans la structure.

Il faut remarquer qu'on peut utiliser récursivement les extensions. Par exemple la définition de la fonction "CAR" correspond à une extension de la classe syntaxique "elem", et admet comme paramètre une dérivation de la même

classe. Donc l'utilisation suivante est valide:

CAR(CAR(CAR(X)))

3.2 Evaluation des extensions.

Dans le chapitre précédent nous avons conclu qu'un système d'extension syntaxique doit être incorporé au compilateur du langage de base. Nous avons vu aussi que toutes les propositions utilisent le processus d'évaluation typique des macros, en remplaçant le texte de l'appel par le texte du corps de la macro. Mais cette méthode entraine deux critiques importantes:

1) L'apparition d'un appel de macro dans le texte source interrompt le processus normal du compilateur. Après vérification de l'appel et son remplacement par le corps de la macro, le compilateur reprend le contrôle, avec ce nouveau texte. Ce processus implique une marche arrière dans le texte à analyser, et pour l'utilisation d'une même macro, on ré-analysera les composants constants du corps, autant de fois qu'il y a d'appels.

2) Le corps de la macro est une chaîne qui doit appartenir au langage étendu défini par l'extension précédente. Mais aucune action n'est

prise pour vérifier cette appartenance lors de la définition de l'extension. Ainsi on perd la possibilité de détecter les erreurs commises par l'utilisateur. Celles-ci ne seront détectées qu'aprés l'évaluation d'un appel et pendant l'analyse que fera le compilateur du texte après substitution.

Nous allons définir une méthode d'évaluation qui évite ces deux problèmes. L'idée de base de notre proposition est d'extraire toute l'information fournie par la définition d'extension au moment de son traitement, et de réaliser toutes les vérifications possibles.

La signification de toute extension est une phrase non-terminale du langage étendu au moment de la définition. Il est alors envisageable d'analyser syntaxiquement cette phrase. Ceci sera une analyse assez particulière à cause de la présence des éléments non-terminaux. Lors du processus normal, l'objectif de l'analyse syntaxique est de vérifier l'appartenance du texte source à un langage, et de construire son arbre syntaxique (les feuilles sont les éléments terminaux qui composent le texte à analyser). Dans notre cas ce processus sera partiellement réalisé, car le texte à analyser n'est pas une phrase terminale. Mais de toute façon une vérification syntaxique peut être faite et l'arbre syntaxique obtenu aura des feuilles correspondant à des non-terminaux de la grammaire, qui sont les paramètres

formels.

Ainsi dans l'exemple déjà présenté:

EXTSYN <exprif> := NULL <elem.1>|-><elem.1>=ADDR(NIL) FINEXT

on va vérifier que la signification est une dérivation valide de la classe syntaxique "exprif", et on obtient pour GSL ,l'arbre de la figure 3.1.

Cette procédure d'analyse implique des modifications de l'algorithme d'analyse syntaxique du compilateur. Dans le paragraphe suivant, et lors de l'explication détaillée de la méthode, nous nous arrêterons sur ce point.

L'application de cette analyse partielle permet de détecter des erreurs au moment de la définition d'extension, et de résoudre ainsi un des problèmes que présentent les macros.

Mais on peut en plus de l'analyse syntaxique, qui fournit l'arbre syntaxique de la signification, réaliser une élaboration partielle de la sémantique, et construire le programme abstrait de la signification. Nous avons déjà presenté l'idée de programme abstrait, dans la description du modèle de Vienne. Dans un programme abstrait on a tous les renseignements sémantiques d'un texte source, et les éléments qui servent à l'analyse syntaxique ont été éliminés. Ce programme abstrait peut être représenté comme

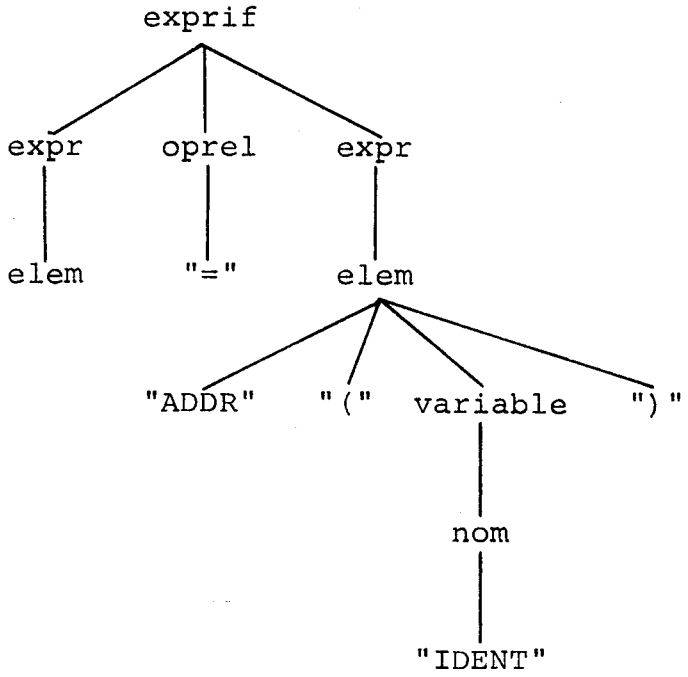


FIGURE 3.1

Analyse syntaxique de la signification.

---

un arbre dont les feuilles correspondent à des actions sémantiques à exécuter afin de compiler le programme source, et il est le résultat de l'application d'une fonction de traduction de l'arbre syntaxique. Dans le cas de la signification le programme abstrait aura des feuilles indéterminées qui correspondent aux paramètres formels de l'extension. Un schéma du programme abstrait de la signification représentée dans la figure 3.1 pourrait être celui de la figure 3.2

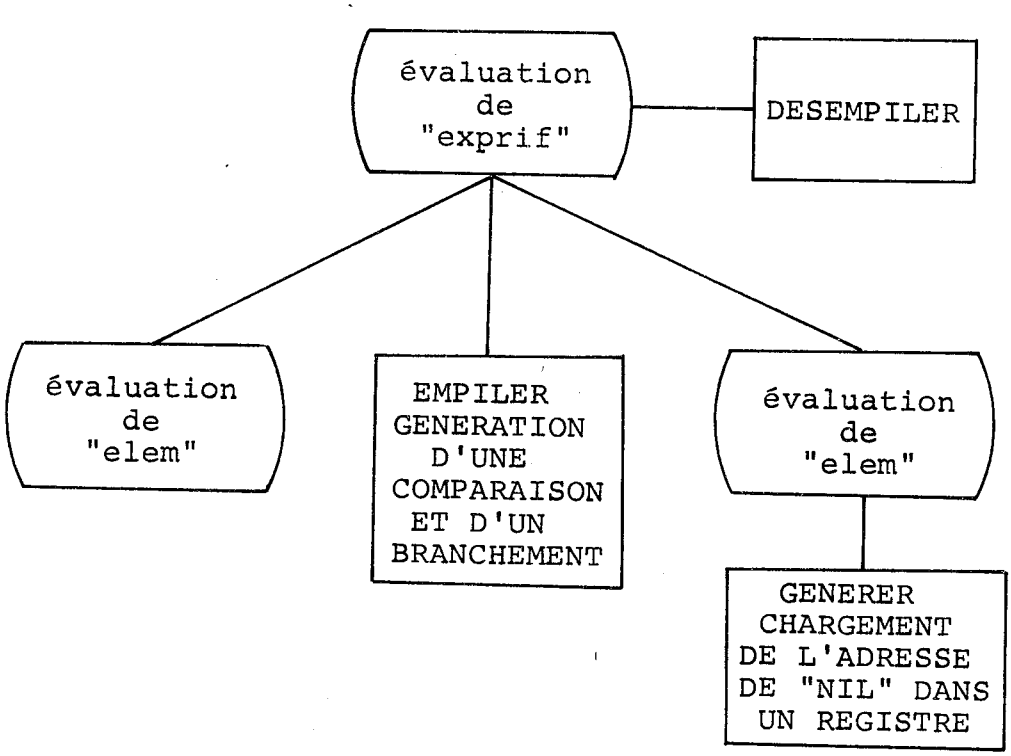


FIGURE 3.2

Programme abstrait de la signification.

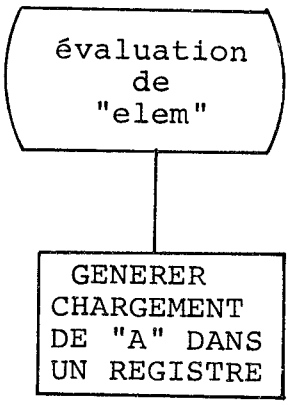


FIGURE 3.3

Programme abstrait du paramètre.

Le résultat de l'exécution d'une définition consiste donc à élaborer le programme abstrait de la signification et à le sauvegarder pour une utilisation ultérieure de l'extension. On peut remarquer ici la différence avec un système de macros, où la valeur à garder est le corps de la macro, tel qu'il a été défini par l'utilisateur.

Lors d'une utilisation les paramètres subiront un traitement similaire à celui de la signification, et on construira des programmes abstraits pour chacun d'eux. Ainsi par exemple lors de l'utilisation:

IF NULL(A) THEN .....

on construira pour le paramètre "A" le programme abstrait de la figure 3.3.

Dès que l'analyse d'une utilisation est terminée, on assemblera tous les programmes abstraits des paramètres réels avec celui de la signification, en formant maintenant un programme abstrait unique pour l'extension. Ainsi suivant l'exemple précédent on attachera le programme abstrait du paramètre "A", à la feuille indéterminée du programme abstrait de la signification qu'on observe dans la figure 3.2.

Le processus d'extension s'achève avec l'interprétation de ce dernier programme, ce qui conduit à l'exécution des actions sémantiques déduites de l'extension.

Ce processus de travail peut se résumer en disant

qu'au moment de la définition d'une extension on avance dans la "compilation" de la signification le plus possible, en repoussant la réalisation d'un processus analogue pour les paramètres au moment de l'utilisation.

On a remplacé ainsi la substitution de textes qu'utilisent les macros par une méthode d'évaluation plus fiable et plus efficace. La fiabilité apparaît comme conséquence des vérifications qu'on réalise à la définition. L'efficacité est atteinte du fait qu'il n'y a aucune analyse qui soit répétée, même pour plusieurs utilisations d'une extension.

Cette méthode d'évaluation, dont nous venons de présenter les idées essentielles, implique une relation très étroite entre le système d'extension et le compilateur du langage de base. Il ne suffit pas, maintenant, de participer à l'analyse syntaxique, le système d'extension doit connaître en plus le processus d'élaboration du compilateur. Nous avons déjà dit que l'absence de formalismes pour décrire la sémantique des langages implique un manque d'uniformité de la part des compilateurs.

Nous utiliserons un schéma de compilation basé sur le modèle de définition des langages, présenté auparavant.

3.2.1 Modèle de compilation.

La présentation du modèle de Vienne pourrait servir de base à une implémentation. Un compilateur ainsi conçu, est constitué essentiellement par les fonctions "ACCEPTER", "TRADUIRE" et l'interpréteur. Le rôle de l'interpréteur ne sera plus d'évaluer mais de générer du code, comme dans un processus de traduction. Dans le modèle nous allons dégager ces trois fonctions.

Le processus de compilation est dirigé par la syntaxe du langage. Le problème qui reste à résoudre est de choisir l'algorithme d'analyse syntaxique à utiliser. Chaque algorithme correspond théoriquement à un automate de reconnaissance attaché à un type de langage. Parmi les possibilités on trouve des algorithmes généraux pour des grammaires hors-contexte (Earley (11)), des algorithmes qui parcourent l'arbre syntaxique de bas en haut pour des langages de précédence (Wirth(39)), et LR(k) (Knuth(21)), et des algorithmes qui parcourent l'arbre syntaxique de haut en bas pour les langages LL(k) (Foster (12), Knuth(21)). Tous ces algorithmes sont utilisés par diverses implémentations.

Nous nous limiterons à l'utilisation d'un algorithme descendant qui reconnaît les langages LL(1). Rappelons brièvement les caractéristiques des grammaires LL(1):

1) Pour toutes les classes syntaxiques, les premiers éléments terminaux des différentes alternatives de génération, doivent appartenir à des ensembles disjoints.

2) Une classe syntaxique ne peut pas générer le mot vide de deux manières différentes.

3) Si une classe syntaxique génère le mot vide, alors elle ne peut pas être suivie par un symbole terminal qui appartient à l'ensemble de ses éléments premiers.

L'analyse syntaxique des langages LL(1) est faite de manière déterministe (sans retour-arrière), en utilisant à chaque instant la prochaine entité lexicographique, pour choisir le chemin à parcourir. Cette méthode est très répandue; ainsi pour la construction automatique des compilateurs on dispose du Transformateur de Grammaires de Griffiths et Peltier, et du "Compiler description language" de Koster(22); et parmi les compilateurs concrets on trouve celui du langage PASCAL (41). L'intérêt de la méthode réside dans l'aspect déterministe de l'analyse syntaxique et dans la facilité d'insérer des appels de fonctions sémantiques dans les règles de production de la grammaire. Grâce à ces caractéristiques le processus de compilation peut être éventuellement réalisé en un seul passage du texte.

Si on compare ce type de compilateur et le schéma déduit du modèle de Vienne on constate que les trois fonctions (ACCEPTER, TRADUIRE et génération du code) sont regroupées en une seule fonction.

3.3 Conception du système d'extension syntaxique.

La définition du système est basée sur un compilateur du langage de base qui suit les critères qu'on vient de décrire. Pour un programme écrit dans le langage de base, la traduction est assurée par le compilateur réalisant sa tâche normale en un seul passage. Par contre lors du traitement des extensions ce processus est coupé en deux parties. La première correspond à la composition des fonctions ACCEPTER et TRADUIRE. Elle sera appliquée à la signification au moment de la définition de l'extension, et aux paramètres réels lors d'une utilisation pour obtenir les programmes abstraits respectifs. Lorsque l'analyse d'une occurrence d'utilisation est terminée on exécutera la deuxième étape, qui consiste à réunir les programmes abstraits obtenus lors de la première étape. Le programme abstrait ainsi construit, sera ensuite interprété afin de générer le code correspondant.

3.3.1 Comportement du système.

Structurellement, le système se compose du compilateur du langage de base et des routines qui apparaissent dans le schèma de la figure 3.4.

On peut distinguer trois états dans lesquels le système peut se trouver à un instant donné: état du langage de base, état de définition des extensions, état d'utilisation des extensions.

3.3.1.1 Etat du langage de base.

Dans cette situation le système se comporte comme si le mécanisme d'extension n'existait pas. Tant que toutes les tentatives vérifiant les éléments syntaxiques du langage de base réussiront, il n'y aura pas d'essai pour trouver des utilisations d'extensions.

3.3.1.2 Etat de définition des extensions.

Un des objectifs que nous nous sommes proposés d'atteindre au début de ce travail était de construire un système le plus général possible. Cela veut dire isoler les primitives du système d'extension des caractéristiques propres du langage de base. Il est évident, et nous l'avons

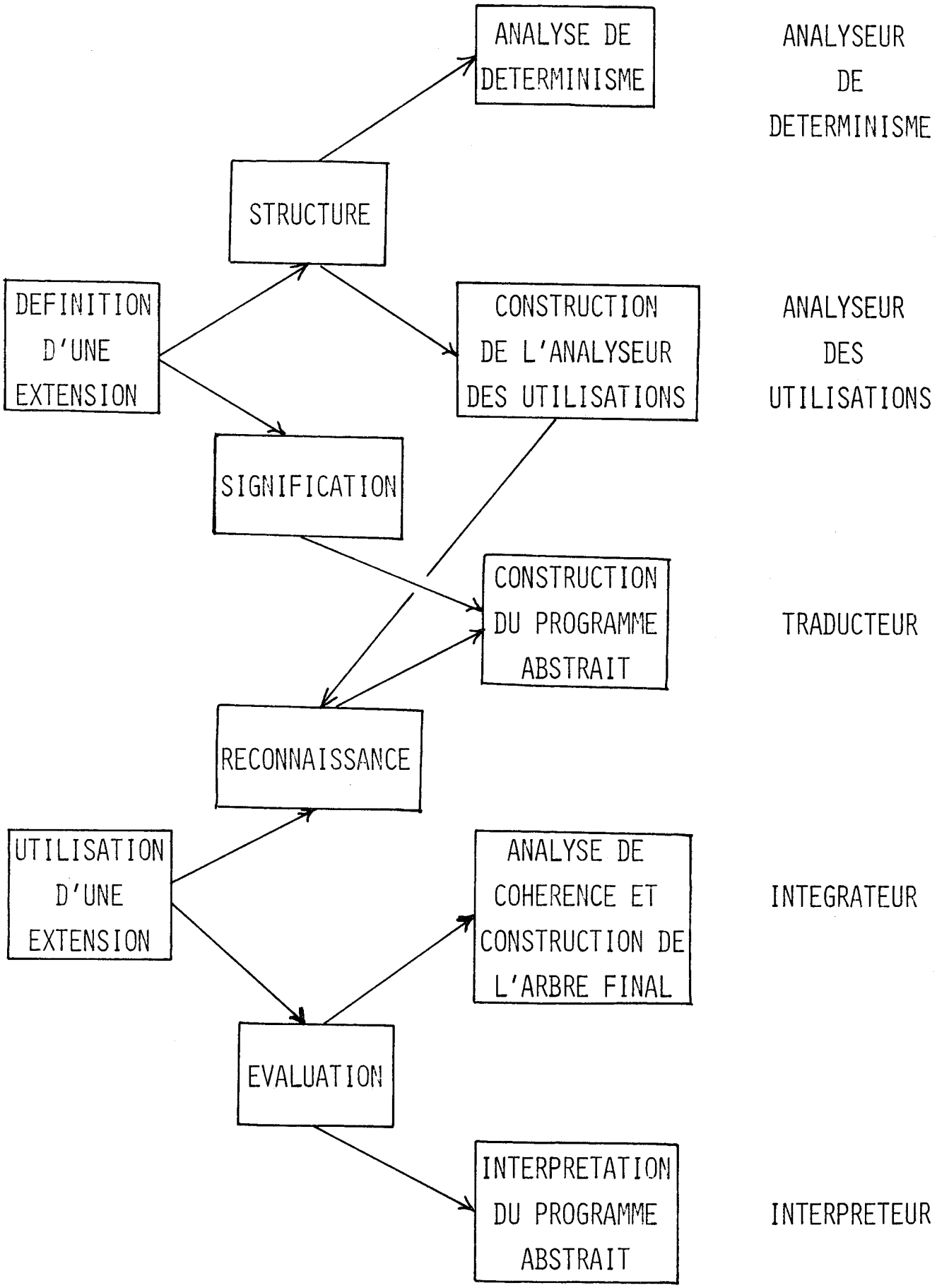

ANALYSE DE DETERMINISME
ANALYSEUR DE DETERMINISME
STRUCTURE
DEFINITION D'UNE EXTENSION
CONSTRUCTION DE L'ANALYSEUR DES UTILISATIONS
ANALYSEUR DES UTILISATIONS
SIGNIFICATION
CONSTRUCTION DU PROGRAMME ABSTRAIT
TRADUCTEUR
RECONNAISSANCE
UTILISATION D'UNE EXTENSION
ANALYSE DE COHERENCE ET CONSTRUCTION DE L'ARBRE FINAL
INTEGRATEUR
EVALUATION
INTERPRETATION DU PROGRAMME ABSTRAIT
INTERPRETEUR

FIGURE 3.4

déjà précisé, que l'implantation d'un tel système nécessite un modèle de compilateur précis. Mais la généralité que nous cherchons se manifeste principalement par rapport à la définition du langage de base. Ainsi la seule modification que nous devons faire à ce langage consiste à permettre des définitions d'extensions. Cela constitue l'unique interface formel entre le langage de base et le système d'extension. Cette modification consiste à ajouter une nouvelle production de génération à la classe syntaxique "instruction", afin de reconnaître une définition d'extension, dont la syntaxe est la suivante:

EXTSYN mtv := structure |-> signification FINEXT

où "mtv", "structure" et "signification", sont de nouvelles classes syntaxiques:

- "mtv" correspond aux noms des classes syntaxiques du langage de base.
- "structure" et "signification", sont des concaténations quelconques des éléments terminaux avec des occurrences de "mtv".

Donc la reconnaissance du début d'une définition est une tâche que nous ajoutons au compilateur du langage de base qui par la suite donnera le contrôle aux routines du système.

3.3.1.2.1 Traitement de la structure.

La structure sera traitée par deux routines du système: l'analyseur des utilisations et l'analyseur de déterminisme.

La structure décrit la syntaxe des utilisations, et par conséquent il faut manipuler cette information afin de détecter, analyser, et vérifier les ocurrences d'utilisation. Cette manipulation consiste à construire un analyseur syntaxique à partir de la structure. Dès que le langage de base est LL(1), il est souhaitable que le langage étendu soit lui aussi LL(1). De cette façon le système reste orthogonal, et on peut utiliser un seul algorithme d'analyse syntaxique. Pour la construction de l'analyseur des extensions nous allons utiliser la procédure définie par Griffiths et Peltier dans leur Transformateur de Grammaires (14). Cette procédure admet comme entrée une grammaire écrite sous la forme BNF, et construit automatiquement l'analyseur syntaxique du langage, après avoir réalisé certaines transformations pour rendre LL(1) la grammaire d'entrée. Le système ne fera aucune transformation de la grammaire qui décrit l'extension; il est de la responsabilité de l'utilisateur de respecter les conditions LL(1) de son langage étendu.

Les conflits possibles de déterminisme introduits lors d'une définition d'extension sont en relation avec les conditions 1) et 3) des langages LL(1) que nous avons décrit auparavant.

Si le premier élément de la structure est, ou peut générer, un élément terminal qui soit "premier" de la classe syntaxique à étendre, la première condition pour les langages LL(1) ne sera plus valide pour le langage étendu. Si en plus la classe syntaxique en question génère le mot-vide dans le langage de base, le premier composant de la structure ne doit pas non plus appartenir à l'ensemble de ces "successeurs". Nous utiliserons la dénomination des "symboles directeurs" pour englober les "premiers" et les "successeurs" d'une classe syntaxique.

Pour éviter ces inconvénients nous imposons que la structure commence par un élément terminal qui ne génère pas de conflit avec la classe syntaxique à étendre. Cette contrainte existe déjà dans la proposition de Leavenworth de macros syntaxiques. Le système vérifiera facilement cette condition au moment de la définition de l'extension.

Dans la structure les classes syntaxiques du langage de base seront concaténées à droite avec des symboles terminaux. Donc on ajoute des "successeurs" à ces classes syntaxiques, et la troisième condition des langages LL(1), peut ne pas être satisfaite si ces nouveaux "successeurs" appartiennent à l'ensemble des "premiers" de la classe syntaxique en question. Voyons un exemple de ce problème. Si

on accepte dans le langage de base qu'une variable puisse être un identificateur, ou un identificateur suivi d'une expression parenthesée, la grammaire LL(1) correspondante peut être écrite:

variable ::= IDENTIFICATEUR mv1

mv1 ::= MOT-VIDE | (expression)

Si dans cette situation on a la définition d'extension:

EXTSYN instruction := ..... variable (.....

alors, la grammaire du langage étendu cesse d'être LL(1), puisque "mv1" a un premier "(", qui est aussi son successeur. On doit donc déterminer pour chaque définition d'extension si les occurrences des symboles terminaux à droite de chaque classe syntaxique conduisent à une telle situation de conflit. Il y a plusieurs manières de résoudre ce problème. En recherchant l'efficacité d'analyse du langage étendu, nous avons défini dans (36), un algorithme qui, utilisant la grammaire du langage de base, détermine pour chaque classe syntaxique la liste de tous les symboles terminaux qui ne peuvent apparaître à sa droite dans une structure de définition. Par suite l'acceptation d'une définition nécessitera seulement la consultation de la table fournie par l'algorithme. Cette vérification est faite dans le système par la fonction ANALYSEUR DE DETERMINISME.

3.3.1.2.2 Traitement de la signification.

Comme nous l'avons dit, nous allons considérer la signification comme une phrase non-terminale du langage étendu et nous allons l'analyser syntaxiquement et construire son arbre abstrait. Cette procédure consiste à réaliser une application partielle du processus de compilation. Le compilateur étant conçu sur la base du modèle décrit précédemment, il exécute les actions sémantiques au fur et à mesure de l'analyse syntaxique. Le travail de la fonction TRADUCTEUR du système d'extension, consiste à "suivre" le compilateur pendant l'analyse de la signification et collecter tous les appels des fonctions sémantiques en construisant ainsi, le programme abstrait, qui sera sauvegardé pour une utilisation ultérieure.

On peut noter ici que le compilateur est appelé d'une manière "procédurale": on lui fournit comme paramètre la classe syntaxique à analyser, et on reprend le contrôle après son exécution.

3.3.1.3 Etat d'utilisation des extensions.

L'utilisation d'une extension revient à programmer dans le langage étendu.

Le compilateur du langage de base, selon le modèle que nous avons décrit, est dirigé par la syntaxe. Cela veut dire qu'il parcourt l'arbre représentatif de la grammaire quand il analyse un texte source. Si au début de l'analyse d'une classe syntaxique le compilateur trouve dans le texte d'entrée l'élément terminal qui caractérise une des extensions de cette même classe syntaxique, il donnera le contrôle au système d'extension. L'environnement de travail du compilateur sera sauvegardé et remis en état à la fin de l'évaluation de l'extension, ce qui assure la continuité du processus de compilation.

3.3.1.3.1 Reconnaisance d'une utilisation.

Le processus de reconnaissance est semblable a celui de compilation. On doit vérifier la syntaxe du texte source et on va construire les arbres abstraits correspondants aux paramètres réels. La première tâche est dirigée par l'analyseur qu'on a construit au moment de la définition(3.3.1.2.1), et la construction des arbres abstraits sera faite de manière analogue a ce qu'on a fait lors du traitement de la signification(3.3.1.2.2).

3.3.1.3.2 Evaluation d'une utilisation.

Dès qu'on a vérifié l'exactitude d'une utilisation on dispose de tous les éléments afin de procéder à l'évaluation. D'une part on a construit l'arbre abstrait de la signification avec des branches indéterminées pour les paramètres. D'autre part, lors de la reconnaissance de l'utilisation, on a fait subir le même traitement aux paramètres réels. Donc on peut construire l'arbre complet de l'extension et procéder a son évaluation.

3.3.1.3.2.1 Construction de l'arbre abstrait final.

"L'INTEGRATEUR" est la routine du système qui se chargera de vérifier que les divers arbres abstraits des paramètres correspondent aux spécifications des feuilles indeterminées de l'arbre abstrait de la signification, en utilisant les renseignements obtenus au moment de la définition. Dès que cette vérification est faite il construira l'arbre final de l'extension.

3.3.1.3.2.2 Interprétation.

L'arbre final de l'utilisation fournit toutes les informations sémantiques de l'extension. "L'INTERPRETEUR" parcourt cet arbre en exécutant les actions selon le processus classique d'interprétation.

Cette description de la dynamique du système fait appaître clairement le découpage du processus de compilation en deux passages pour l'évaluation des extensions. La première phase commence au moment de l'analyse et du traitement de la signification. Elle est complètée par l'application de la même procédure aux paramètres réels. La deuxième phase est réalisée par l'interpréteur qui exécute les actions sémantiques. Ce comportement du système assure l'efficacité d'exécution qui était un des objectifs de notre travail.

C H A P I T R E I V .

LE COMPILATEUR DU LANGAGE DE BASE.

La définition du système d'extension est basée sur un modèle de compilateur que nous avons décrit dans le chapitre précédent. Ceci permet d'appliquer le système à tout langage dont le compilateur vérifie les conditions du modèle. Nous allons présenter dans ce chapitre une méthode de construction de compilateurs qui utilise le Transformateur de Grammaires développé par Griffiths et Peltier (14), qui fournit automatiquement l'analyseur syntaxique d'un langage à partir de sa grammaire. Cet analyseur est constitué d'une suite de fonctions; nous avons modifié cet aspect de manière à rendre plus aisée la manipulation des extensions.

4.1 Construction du compilateur.

Les grammaires de type LL(1) définissent une classe de langages dont l'analyse syntaxique peut être faite de manière déterministe. Le processus consiste à analyser la chaîne d'entrée de gauche à droite (sans retour arrière), en cherchant d'abord les dérivations les plus à gauche, en fonction de l'entité lexicographique courante.

L'algorithme d'analyse étant parfaitement déterminé, cela permet d'envisager la construction automatique de

l'analyseur syntaxique.

Utilisant cette approche, Griffiths et Peltier (14) ont conçu un système d'aide à la construction de compilateurs. Celui-ci consiste en:

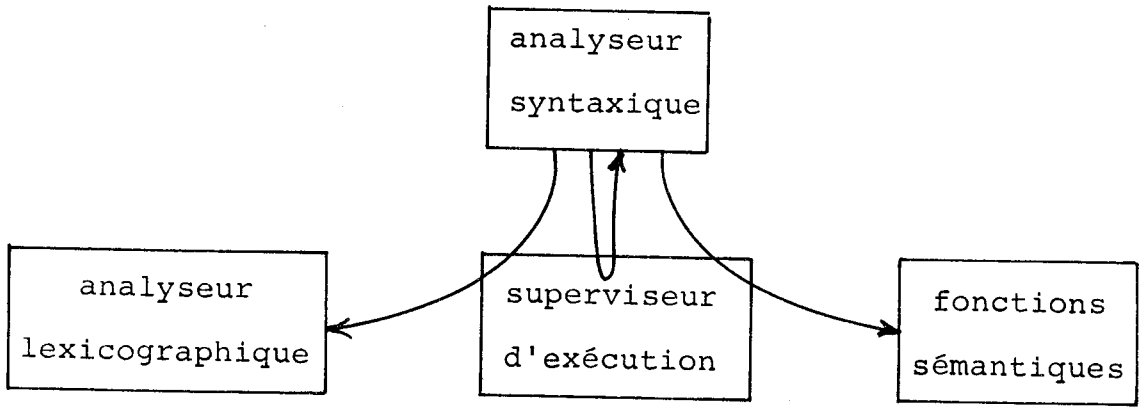
1) un programme qui acceptant une grammaire sous forme BNF:
   - Vérifie les conditions LL(1).
   - Eventuellement, fait quelques transformations pour la rendre LL(1) (factorisation des règles).
   - Produit l'analyseur syntaxique correspondant à la grammaire.

2) un macro-langage qui est utilisé pour l'écriture de l'analyseur, et qui sert également à l'écriture des fonctions sémantiques dont les appels ont été insérés dans la grammaire.

Le système comprend aussi un superviseur d'exécution chargé de gérer la récursivité du processus de compilation.

Un compilateur conçu à l'aide du système précédent

peut être schematisé de la manière suivante:



Nous ne nous arrêterons pas sur les transformations de la grammaire ni sur l'algorithme de construction de l'analyseur, mais nous nous intéresserons à la structure du programme de l'analyseur.

A toute classe syntaxique de la grammaire est associée une routine de l'analyseur syntaxique. Pour chaque alternative de production les occurrences de classes syntaxiques sont des appels aux routines correspondantes. Les éléments terminaux provoquent des appels au pré-processeur afin de vérifier leurs présence dans le texte d'entrée. Le choix parmi les différentes alternatives est fait en fonction des éléments "premiers" obtenus pour la classe syntaxique lors de l'analyse de déterminisme.

Ainsi pour la règle de production:

A -> a B | c D

on obtiendra la routine suivante:

```
        ROUTINE A
        DECIDE L1,SINGLE,a
        CHECK (c)
        CALL D
        EXIT
    L1  CHECK (a)
        CALL B
        RETURN
```

L'analyse syntaxique, déterministe et descendante, permet d'insérer dans les règles de production de la grammaire les noms des actions sémantiques. Lors de la construction de l'analyseur syntaxique ces noms représenteront dans les routines, les appels des fonctions sémantiques correspondantes. Par exemple pour la règle:

A -> a P B

où P est le nom d'une action sémantique, on obtiendra la routine suivante:

```
ROUTINE A
CHECK (a)
CALL P
CALL B
RETURN
```

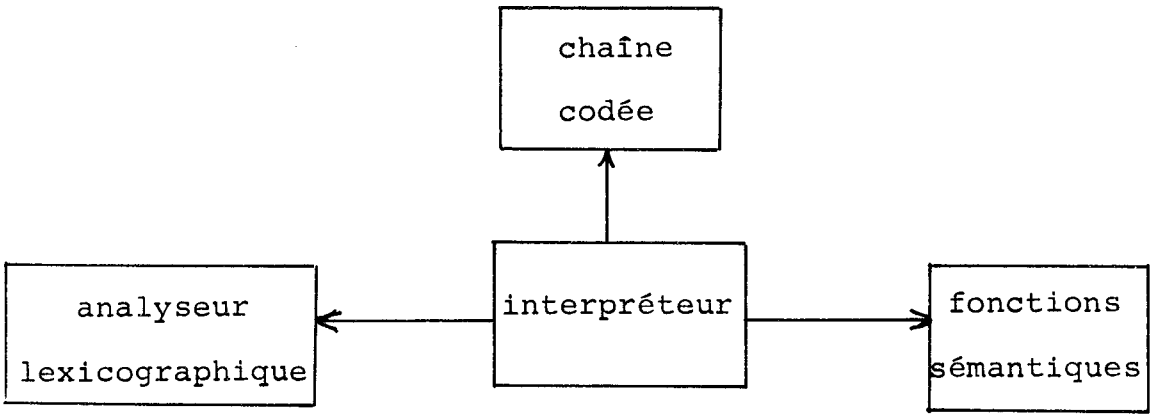
Ceci veut dire que pendant l'analyse de la classe syntaxique "A", on exécutera l'action sémantique "P".

Pour plus de détails concernant le macro-langage se référer à (30).

4.2 Solution interprétative.

Parmi les composants du système d'extension qui doivent être en relation avec le compilateur du langage de base, on a notamment le constructeur de l'analyseur syntaxique des utilisations, et le constructeur des programmes abstraits. Afin que les actions de ces deux fonctions soient identiques aux fonctions analogues du compilateur, nous avons modifié la structure de ce dernier. La modification consiste essentiellement à adopter une solution interprétative (1) pour l'analyseur syntaxique. Ainsi, la sortie du Transformateur de Grammaires est traduite en une chaîne codée, les différents appels des fonctions syntaxiques et sémantiques étant maintenant interprétés au lieu d'être directement exécutés.

Le schéma de la nouvelle structure est le suivant:



Il faut remarquer qu'on laisse aussi à l'interpréteur la gestion de la récursivité du processus de compilation.

Cet aspect interprétatif ne modifie pas la conception globale du compilateur, mais offre la possibilité d'accéder facilement à l'information disponible dans toutes les étapes de la traduction, facilité procurée par le fait que le processus de compilation est maintenant entièrement sous le contrôle de l'interpréteur.

4.2.1 Contrôle du processus de compilation.

En ce qui concerne le système d'extension, le fait d'interpréter l'analyseur syntaxique offre les possiblités suivantes:

1) L'analyseur syntaxique, étant représenté par une chaîne interprétable, peut être modifié dynamiquement lors du traitement d'une définition d'extension. Ainsi, l'analyseur d'une utilisation pourra être généré comme une chaîne dont la structure est semblable à celle de l'analyseur du langage de base. Et, qui plus est, on pourra attacher facilement ce nouvel analyseur à la chaîne du langage de base, rendant ainsi l'analyseur syntaxique du langage étendu tout à fait homogène.

2) En analysant les éléments de la chaîne constituant l'analyseur syntaxique, on peut distinguer facilement les appels aux fonctions sémantiques. Ce fait sera utilisé par le système d'extension, qui pourra ainsi construire les programmes abstraits. Un programme abstrait sera donc représenté comme une structure arborescente dont les feuilles correspondront aux pseudo-instructions interprétables. Ceci permet d'évaluer le programme abstrait de l'extension, au moyen de l'interpréteur qui dirige le processus normal de compilation.

4.2.2 Réalisation de la solution inteprétative.

Afin d'obtenir un compilateur selon le schéma interprétatif nous avons programmé avec P. Jacquet (18) un traducteur, qui à partir d'un analyseur écrit sous forme d'appels de macros produit la chaîne codée correspondante. On appellera, par la suite, pseudo-instructions les composants de cette chaîne.

La syntaxe des pseudo-instructions est:

pseudo-opérateur opérandes

La correspondance entre les macro-instructions et les pseudo-instructions est la suivante:

ROUTINE

Cette macro qui correspond à l'en-tête des routines dans le système primitif, n'a pas de sens dans le processus d'interprétation. Par contre ce qui nous intéresse c'est son adresse relative, qui sera utilisée lors de la traduction des appels aux fonctions syntaxiques.

CALL

On doit distinguer pour des appels, si la fonction associée correspond à une routine

syntaxique ou s'il s'agit d'une fonction sémantique, car ceci entraine des traductions différentes. Un appel d'une routine syntaxique aura comme opérande une adresse dans la chaîne codée, tandis que l'appel à une routine sémantique implique un lien extérieur à cette chaîne. Elles auront donc des pseudo-opérateurs différents.

ENTER

Le problème est analogue a celui de CALL; donc la traduction des ENTER génère des pseudo-instructions différentes pour distinguer les fonctions syntaxiques des actions sémantiques.

Pour les autres macro-instructions, il n'y a pas de traitement spécial, et la sémantique des pseudo-instructions est analogue à celle des macro-instructions.

Le programme de traduction vers une chaîne interprétable, prévoit également que l'analyseur d'entrée ne soit pas la sortie directe du Transformateur de Grammaires, mais une version optimisée de cette dernière.

Ces optimisations automatiques sont réalisées par un post-processeur (3), et elles concernent aussi bien l'occupation mémoire que la vitesse d'exécution.

Nous avons conservé ces optimisations dans la solution interprétative et les modifications qui nous intéressent sont les suivantes:

1) La récursivité à droite dans une règle de production conduit à ce que la routine associée se termine par un ENTER, dont le paramètre est la routine elle même. Ceci est équivalent à un branchement vers le début de la routine.

2) Lors du choix des "premiers" fait par la macro DECIDE, on doit comparer son (ou ses) paramètres avec l'entité lexicographique courante. Si la comparaison est vérifiée on exécute un branchement vers l'instruction dont l'étiquette est le second paramètre du DECIDE. Dans la version générée par le Transformateur de Grammaires cette dernière instruction peut être un CHECK, qui sémantiquement correspond à une comparaison et à un appel au pré-processeur. On peut donc éviter cette double comparaison en substituant le CHECK par l'appel au pré-processeur afin de mettre à jour l'entité lexicographique courante.
Il peut arriver qu'on trouve une suite d'appels au pré-processeur, correspondants à des DECIDE différents. Dans ce cas, le processus

l ensemble des étiquettes associées.

Voyons ceci plus clairement au moyen d'un exemple: la règle de production:

A->a A | b | c

conduit aux situations suivantes:

version non-optimisée

```
   ROUTINE A
   DECIDE L1,SINGLE,b
   DECIDE L2,SINGLE,c
   CHECK (a)
   ENTER A
L1 CHECK (b)
   EXIT
L2 CHECK (c)
   RETURN
```

version optimisée

```
   ROUTINE A
   DECIDE L1,SINGLE,b
   DECIDE L2,SINGLE,c
   CHECK (a)
   B  A
L1 EQU *
L2 EQU *
   CALL PREPROC
   RETURN
```

Le processus d'optimisation introduit ainsi des macro-instructions qui vont être traduites en deux nouvelles pseudo-instructions de la chaîne codée.

Le détail des pseudo-instructions apparaît dans l'annexe I ainsi qu'une description de l'algorithme de l'interpréteur.

La solution interprétative a également un autre avantage. Elle libère l'écriture du compilateur d'un langage particulier. La chaîne codée, générée automatiquement, peut être la même pour des interpréteurs écrits dans des langages de programmation différents. Ceci donne un degré de liberté suplémentaire à la construction du compilateur qui se manifestera également dans le système d'extension.

Enfin nous pouvons comparer les performances des deux solutions. A ce sujet soulignons que le compilateur basé sur l'interprétation de la chaîne codée occupe moins de place que la version initiale. Quant au temps de compilation il est superieure de seulement 10% pour des programmes moyens (200 lignes GSL).

C H A P I T R E V .

UNE REALISATION DU SYSTEME D'EXTENSION.

Nous allons présenter maintenant une réalisation du système d'extension syntaxique. L'objectif à atteindre avec cette implémentation consiste à vérifier les algorithmes que nous avons décrits précédemment, surtout en ce qui concerne l'évaluation des extensions.

Dès l'origine de notre travail, nous nous sommes proposés de réaliser un système indépendant d'un langage particulier et ayant un champ d'application très vaste. Pour cela nous avons basé la conception du système sur un modèle de compilateur du langage de base. Lors de la présentation que nous allons faire nous essayerons de dégager clairement les caractéristiques du système et les modifications qu'on doit faire subir au langage de base; nous verrons que ces dernières sont minimales.

Cette présentation doit servir également comme exemple d'application du système à d'autres langages dont le compilateur est conforme au modèle; une telle réalisation pourra être faite dans une période très courte par une personne qui connaît le langage.

Pour cette réalisation nous avons choisi GSL (4) comme langage de base. Cette décision a été prise parce que nous disposons d'un compilateur GSL qui réunit les caractéristiques du modèle que nous avons proposé.

L'analyseur syntaxique a été obtenu à l'aide du Transformateur de Grammaires, et les fonctions sémantiques ont été écrites dans le même macro-langage que l'analyseur.

Afin d'incorporer le système d'extension syntaxique il nous fallait disposer du compilateur dans une version qui interprète l'analyseur syntaxique.
Pour ce faire nous avons dû réaliser au préalable les modifications ou travaux suivants:

a) Faire subir à l'analyseur syntaxique le processus décrit dans le chapitre précédent afin d'obtenir la chaîne interprétable. Le schéma de ce processus est présenté dans la figure 5.1.

b) Ecrire l'interpréteur de cette chaîne. Pour ce programme nous avons utilisé GSL. Ce dernier étant un langage d'écriture de systèmes s'adapte très bien à nos besoins.

c) De façon à utiliser les fonctions sémantiques du compilateur de départ nous avons programmé un interface entre l'interpréteur, écrit en GSL, et les fonctions sémantiques écrites dans le macro-langage. Cette routine simule le superviseur d'exécution de la première version du compilateur pour les fonctions sémantiques.

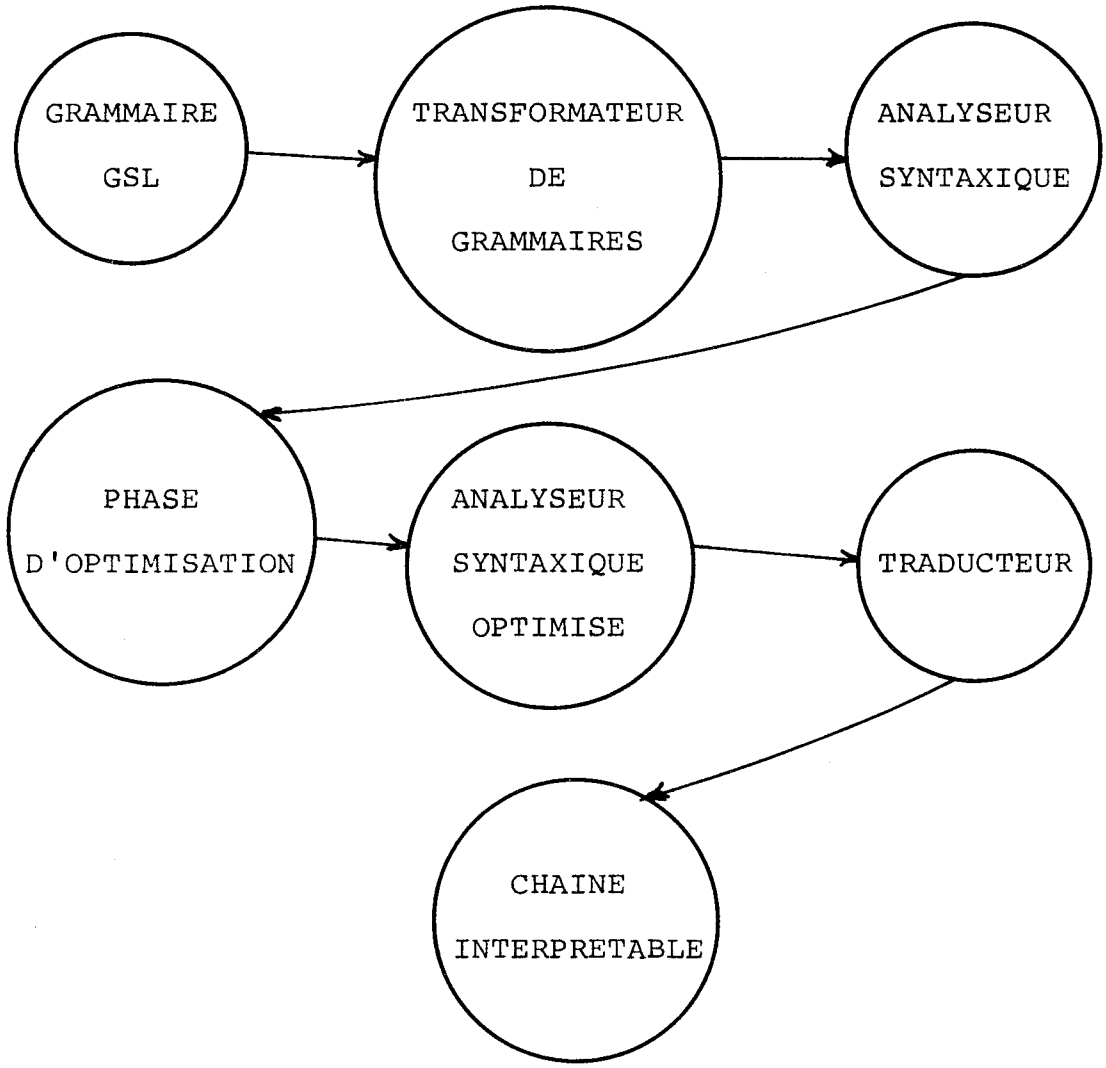

GRAMMAIRE
GSL
TRANSFORMATEUR
DE
GRAMMAIRES
ANALYSEUR
SYNTAXIQUE
PHASE
D'OPTIMISATION
ANALYSEUR
SYNTAXIQUE
OPTIMISE
TRADUCTEUR
CHAINE
INTERPRETABLE

FIGURE 5.1

Cette dernière modification n'est évidemment pas nécessaire dans le cas où le compilateur est conçu selon la méthode interprétative. Dans cette situation l'interpréteur et les fonctions sémantiques seront écrits dans un même langage, et les relations entre ces deux parties seront faites par l'éditeur de liens du système d'exploitation.

Nous avons déjà dit qu'afin de pouvoir appliquer le système d'extension au plus grand nombre de langages, les modifications à apporter lors de la définition de ces derniers doivent être minimales. Essentiellement nous devons ajouter la possibilité de définir des extensions. Ceci implique l'apparition de nouvelles règles de production dans la grammaire. Tout le processus de reconnaissance comme celui d'évaluation des utilisations reste indépendant de la définition du langage de base. Pour cette réalisation dans le système d'extension nous avons conçu les fonctions chargées du traitement des définitions, de manière analogue aux fonctions sémantiques du compilateur. Cela veut dire que leurs activations seront faites pendant le processus d'analyse syntaxique. On pourra dire que ceci constitue également des modifications à la définition du langage de base, du moment qu'il s'agit de la sémantique des définitions d'extension. Mais le contenu de ces nouvelles fonctions concerne essentiellement le système d'extension et

reste indépendant d'un langage particulier.

Pour la présentation qui suit nous distinguerons les composants du système que nous avons considéré comme des fonctions sémantiques des modifications à l'interpréteur du compilateur.

5.1 Modifications syntaxiques.

Nous devons ajouter à la grammaire du langage de base les règles adéquates afin de pouvoir définir des extensions. Rappelons que la syntaxe d'une définition est la suivante:

```
EXTSYN  classe  :=  structure |->  signification   FINEXT
```

où:

classe est le nom de la classe syntaxique objet de l'extension.

structure est une concaténation quelconque d'éléments terminaux et non-terminaux. Ces derniers correspondent aux paramètres formels de l'extension. Le premier composant de la structure doit être un terminal.

signification est une phrase qui comporte des non-terminaux du langage étendu existant au moment de la définition.

Pour la grammaire du langage GSL nous avons choisi la classe syntaxique "sta" comme racine pour les définitions. Cette classe est celle qui génère les instructions du langage.

Les règles qu'on ajoute à la grammaire sont les suivantes:

```
sta := extsyntx
extsyntx := EXTSYN  regleext  FINEXT
regleext := structure '|->'
structure := '<' IDE  '>' ':=' terminal  liselestr
terminal := IDE
liselestr := terminal  liselestr |
             symbol  liselestr |
             mtv liselestr |
             MOT-VIDE
mtv := '<'  IDE  '>.' INTEGER
```

Remarques:

- On représente en majuscules les éléments terminaux du langage. EXTSYN et FINEXT deviennent des mot-clés.
- IDE et INTEGER sont des éléments lexicographiques correspondant respectivement aux ensembles des identificateurs et des entiers. La distinction pour chaque élément de

ces ensembles est faite au niveau du pré-processeur.

- "symbol" est une classe syntaxique qui peut produire les symboles définis dans le langage de base.
- La signification de la définition n'apparaît pas explicitement dans la syntaxe. Du moment qu'elle subira une compilation "partielle", son traitement implique un appel récursif au compilateur, comme nous le verrons dans ce qui suit.

5.2 Fonctions sémantiques.

Les définitions d'extension, dont nous venons de décrire la syntaxe, impliquent des actions sémantiques à accomplir. Les actions correspondent :

- à la construction de l'analyseur syntaxique de l'extension,
- à l'analyse de déterminisme du langage étendu,
- au traitement de la signification de la définition,
- et à la manipulation des paramètres formels.

L'analyse syntaxique, descendante et déterministe, nous permet de décomposer ces actions en fonctions

sémantiques, dont les appels seront insérés dans la syntaxe. Par suite, chacune des fonctions est parfaitement déterminée, et l'endroit de son activation est connu.

5.2.1 Analyseur des extensions.

La construction de l'analyseur des extensions est une version simplifiée du processus accompli par le Transformateur de Grammaires: à partir d'une grammaire, représentée par la structure, on obtient l'analyseur syntaxique. Mais on ne réalise aucune transformation et la grammaire est constituée par une seule règle de production.

Avant de construire l'analyseur on doit exécuter un processus d'initialisation. Ce processus inclut l'acquisition de mémoire pour l'analyseur et fondamentalement il établit les liens avec l'analyseur du langage de base afin de fournir un analyseur homogène pour le langage étendu. Une telle situation a l'avantage de permettre l'utilisation d'un seul interpréteur pour les extensions et pour le langage de base.

5.2.1.1 Initialisation de l'analyseur.

Une occurrence d'utilisation doit commencer par l'élément terminal associé à l'extension. La première action de l'analyseur sera donc de vérifier la présence de cet élément dans le texte source. Les situations analogues dans l'analyseur du langage de base sont résolues au moyen de l'instruction "DECIDE". Dans ce cas, où le paramètre de cette instruction est une entité lexicographique, on ne peut pas appliquer la même procédure (le terminal est en fait un identificateur particulier). Ceci nous oblige à définir une nouvelle pseudo-instruction dont la sémantique diffère de celle de l'instruction DECIDE par le test sur un identificateur particulier et non plus sur une entité lexicographique. Nous utilisons le fait que le pré-processeur du compilateur GSL fournit pour les identificateurs une valeur qui est un pointeur vers une entrée dans la table des identificateurs. Cette valeur, étant en relation biunivoque avec l'identificateur, sera utilisée pour le représenter.

La syntaxe de la nouvelle pseudo-instruction est la suivante:

code ident pointeur nombre

où:

code est la nouvelle pseudo-opération (nous avons pris le numéro 14).

ident est la valeur représentative de l'identificateur associé à l'extension.

pointeur est une adresse dans la chaîne codée où se brancher en cas d'échec lors du test de l'identificateur.

nombre correspond au nombre de paramètres de l'extension.

Un de nos objectifs de base consiste à utiliser le maximum de fonctions du compilateur dans le système d'extension. En ce qui concerne l'analyseur des extensions, le but est de créer une structure cohérente avec l'analyseur du langage de base, de façon à pouvoir utiliser le même processus d'interprétation pour le langage étendu. Ainsi la zone réservée par le compilateur pour la chaîne codée doit être élargie afin de pouvoir contenir les analyseurs d'extensions. Le lien entre l'analyseur existant et le nouveau sera établi par l'intermédiaire de la pseudo-instruction qu'on vient de présenter. Le problème consiste à attacher une nouvelle branche à un arbre donné. La chaîne codée représente l'arbre de la grammaire du langage de base; une extension correspond à une nouvelle règle de production pour une classe syntaxique. Il faut donc, connaître l'adresse dans la chaîne codée où commence la pseudo-instruction associée à la classe syntaxique objet

de l'extension, et réaliser dans cette zone l'association correspondante avec l'analyseur de l'extension.

Le Transformateur de Grammaires représente les classes syntaxiques au moyen d'un numéro. Pour des raisons de lisibilité et afin de faciliter l'utilisation du système, les références aux classes syntaxiques seront faites par les mêmes noms qui apparaissent dans la grammaire de définition du langage. Ceci nous a amené à réaliser une sortie suplémentaire du Transformateur de Grammaire en forme de table de relation entre les noms des classes syntaxiques et les numéros correspondants. Cette table est lue au moment de l'initialisation du compilateur et sera utilisée lors de chaque référence à une classe syntaxique.

Nous disposons également d'une table de pointeurs, qui indique pour chaque classe syntaxique, l'adresse dans la chaîne codée de l'analyseur où commence la pseudo-routine associée. Cette table a été construite par le traducteur vers la chaîne codée et elle est aussi lue au moment de l'initialisation du compilateur.

Lors de l'analyse d'une définition, et au moment où l'on reconnaît le nom de la classe syntaxique à étendre, on peut accéder à l'emplacement dans la chaîne où se trouve la routine associée. Ceci est fait en utilisant séquentiellement les deux tables qu'on vient de décrire.

Pour l'insertion de l'analyseur d'extensions il existe deux situation à considérer:

A) Supposons d'abord qu'il n'y ait pas de définition d'extension préalable pour la classe syntaxique. Le processus d'incorporation du nouvel analyseur est le suivant:

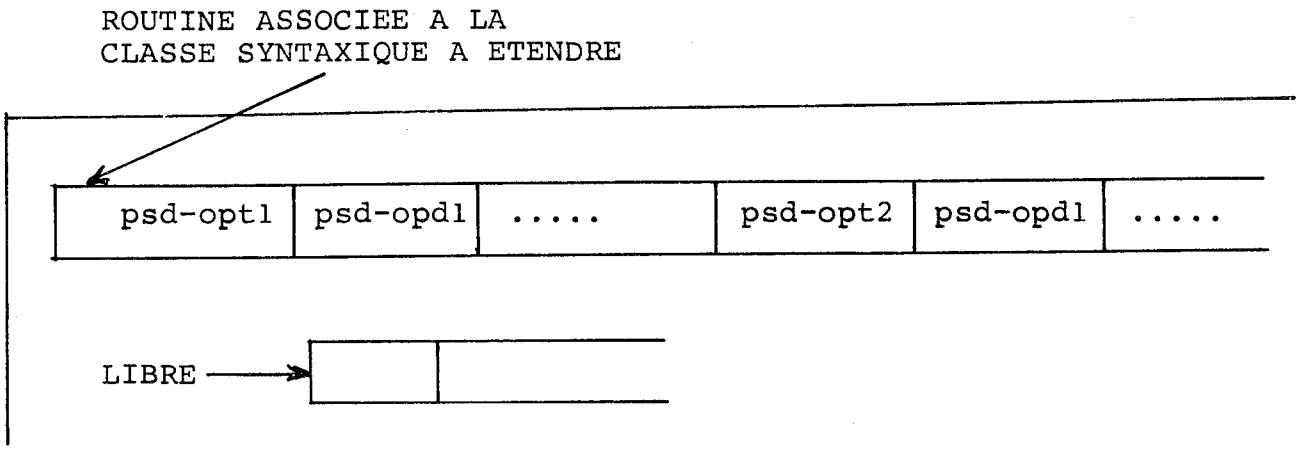
a) On recopie dans la zone libre de la chaîne codée la première pseudo-instruction de la routine, suivie d'un branchement vers la deuxième pseudo-instruction.

b) On initialise l'analyseur de l'extension avec la nouvelle pseudo-instruction qu'on a décrite précédement. Son paramètre "pointeur" a l'adresse du premier composant de la recopie.

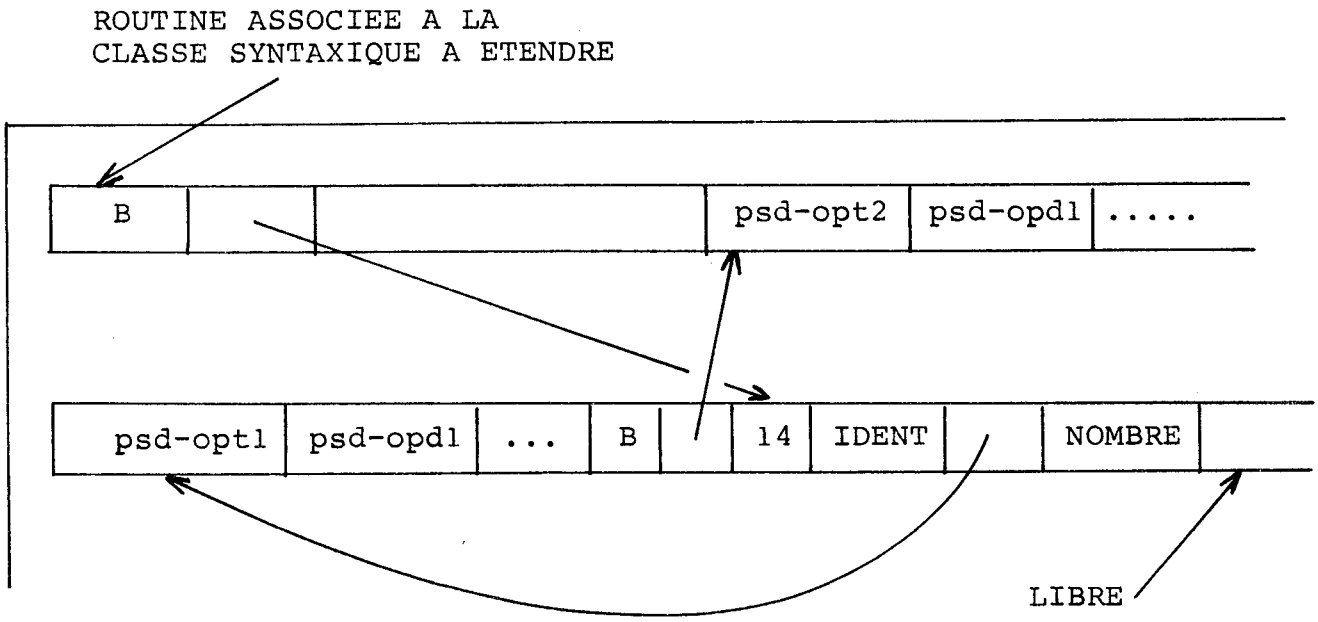
c) On remplace la première pseudo-instruction de la routine par un branchement vers la tête de l'analyseur de l'extension.

On peut suivre ce processus avec le diagramme de la figure 5.2.

Le processus de recopie doit tenir compte de la longueur de la pseudo-instruction à substituer sinon on risque d'avoir une perte d'espace considérable. Le cas le



AVANT LA DEFINITION



APRES LA DEFINITION

FIGURE 5.2

plus défavorable est celui de la pseudo-instruction DECIDE car le nombre des paramètres n'est pas déterminé à priori. Ceci provient du fait que le Transformateur de Grammaires englobe dans un seul DECIDE toutes les comparaisons pour les "premiers" d'une alternative. Pendant la recopie on détecte une telle situation, et on découpe le DECIDE "long" en deux: un pour le premier test et qui sera recopié dans la zone libre, et un deuxième pour les autres cas qui reste dans l'emplacement initial.

Avec une telle conception le processus d'interprétation est le suivant: étant dans le contexte de la classe syntaxique on essaye premièrement de déterminer si on est en présence d'une utilisation d'extension. Si c'est le cas on analyse l'extension; sinon on exécute le processus normal du langage de base en utilisant la recopie de la première pseudo-instruction; le branchement qui le suit nous conduit à la chaîne originale. Evidemment, ceci est une décision d'implémentation qui pourrait être différente pour d'autres applications. Ainsi on peut imaginer les tests pour les extensions seulement après avoir essayé toutes les structures du langage de base. Cette solution permet de tester en dernier les situations les moins probables. Nous avons cependant adopté la solution inverse, ceci en raison du fait que l'élément "premier" d'une extension est un identificateur. Cela peut conduire à une situation ambigüe,

si la classe syntaxique objet de l'extension, a comme "premier" dans le langage de base, l'ensemble des identificateurs. Ce problème n'apparaîtrait pas dans une réalisation où on pourrait modifier dynamiquement la table des mots-clés de l'analyseur lexicographique.

On a obtenu ainsi un analyseur homogène pour le langage étendu avec seulement la définition d'une instruction nouvelle. Ceci, évidement, doit être acompagné d'une modification de l'interpréteur.

B) La seconde situation se présente si au moment de la définition il existe déjà des extensions pour la classe syntaxique, l'analyseur aura la forme de la deuxième partie de la figure 5.2. L'addition du nouvel analyseur est faite en fonction du critère suivant: la dernière extension définie sera la première à être interprétée. Dans la zone libre on crée une nouvelle instruction d'en-tête de l'analyseur. Pour la mise à jour des pointeurs on procède de la manière suivante: le paramètre du branchement où commence la routine pointera vers la nouvelle instruction; le paramètre de cette dernière prendra la valeur initiale du branchement.

Les conflits possibles avec les conditions de déterminisme, que peut introduire une nouvelle extension

seront analysés dans le paragraphe suivant.

Pour la réalisation en GSL nous avons inclu tout le processus d'initialisation dans une fonction sémantique, dont l'appel sera fait pendant l'analyse d'une définition après reconnaissance du nom de la classe syntaxique à étendre.

5.2.1.2 Construction de l'analyseur.

La construction du corps de l'analyseur est réalisée au fur et à mesure qu'on analyse la structure d'une définition. Pour chacun des composants on aura une pseudo-instruction générée. Dans une structure il peut y avoir trois types différents d'éléments: symboles spéciaux, identificateurs, paramètres formels.

a) Symboles spéciaux.

Le traitement des symboles spéciaux est analogue à celui réalisé par le Transformateur de Grammaires. On génère un test au moyen de la pseudo-instruction CHECK, avec comme paramètre l'entité lexicographique associée au symbole.

b) Identificateurs.

Pour les identificateurs nous devons introduire une nouvelle pseudo-instruction. Du point de vue de la sémantique elle correspond à un CHECK, mais son paramètre ne sera pas l'entité lexicographique commune aux identificateurs, mais la valeur de son entrée dans la table des identificateurs. Nous avons vu que ce pointeur est mis par le pré-processeur dans une variable globale.

c) Paramètres formels.

Les traitements que doivent subir les paramètres réels lors d'une utilisation sont: analyse syntaxique et construction du programme abstrait. Afin de réaliser ce processus on utilise les fonctions du compilateur du langage de base. Ainsi, l'occurrence d'un paramètre formel dans la structure implique l'appel à la routine du compilateur dont le nom apparaît explicitement dans le paramètre. Rappelons que la syntaxe d'un paramètre est:

<classe-syntaxique> . numéro

et que "numéro" sert à distinguer les différents paramètres.

Toujours en cherchant une structure cohérente pour le langage étendu, les programmes abstraits des paramètres réels seront eux aussi alloués dans la zone où se trouve la chaîne codée du langage de base. Cette allocation est temporaire car la "vie" de ces programmes est éphémère, et leur durée correspond à l'évaluation d'une utilisation.

Les actions entreprises par le constructeur de l'analyseur sont les suivantes:

- L'appel à la routine du compilateur est fait sous le contrôle d'une fonction ad-hoc. Donc, lorsqu'on reconnaît le nom de la classe syntaxique du paramètre on génère dans l'analyseur un appel à cette fonction. Cet appel contient aussi le nom de la routine syntaxique et l'adresse où on doit placer le programme abstrait.
- Lorsqu'on identifie le paramètre avec le numéro associé, on fait l'acquisition de deux emplacements dans la zone prévue pour le programme abstrait du paramètre. Dans le premier on met le nom de la routine syntaxique, et dans le deuxième le numéro qu'identifie le paramètre. Cette information sera utilisée lors du traitement de la signification.

5.2.2 Déterminisme du langage étendu.

Les vérifications à réaliser pour la structure d'une définition afin d'analyser les conditions de déterminisme du langage étendu sont:

A) Pour le premier composant de la structure on doit vérifier qu'il n'appartient pas à l'ensemble des "symboles directeurs" de la classe syntaxique à étendre. Pour ce faire le système dispose d'une table qui fournit la suite des "symboles directeurs" pour chaque classe syntaxique, et la vérification de déterminisme se limite à une consultation de cette table. On doit vérifier également qu'on n'introduit pas de conflits avec le langage étendu. Cela veut dire qu'il n'y a pas d'extension préalable qui comporte le même premier composant. Pour ce faire nous disposons d'une table qui donne pour chaque classe syntaxique les premiers composants des extensions existantes.

B) Chaque classe syntaxique qui apparaît dans la structure peut conduire à une situation de conflit avec les conditions de déterminisme du langage étendu. Nous avons déjà analysé ce problème dans le chapitre 3. Nous avons défini un algorithme (36) qui nous fournit l'information nécessaire afin de tester les caractéristiques LL(1) du langage étendu. Le programme qui correspond à cet algorithme

utilise comme entrée des informations sur le langage de base qui sont obtenues à partir du Transformateur de Grammaires modifié afin d'avoir les trois éléments suivants:

i) Liste de toutes les classes syntaxiques qui génèrent le mot-vide.

ii) Liste de toutes les classes syntaxiques avec les "symboles directeurs" associés.

iii) Grammaire déterministe.

La sortie de ce programme est sous la forme d'une liste associant à chaque classe syntaxique l'ensemble des éléments terminaux ne pouvant pas apparaître à sa droite dans la structure d'une définition. Cette liste est lue par le compilateur au moment de son initialisation et consultée à chaque occurrence d'une classe syntaxique dans la structure.

5.2.3 Traitement de la signification.

Le traitement de la signification implique la construction de son programme abstrait en utilisant le compilateur du langage de base modifié par les définitions d'extension préalables. Un programme abstrait sera constitué par une suite d'appels à des fonctions sémantiques et d'informations sur l'environnement de travail de ces

fonctions. Le compilateur dispose d'une zone de variables globales servant à la communication entre les différentes routines (caractère lexicographique courant, renseignements sur les identificateurs et sur les constantes, etc). Ces informations doivent être sauvées lors de la construction du programme abstrait de façon à ce que pendant le processus d'interprétation on puisse donner à chaque fonction l'environnement nécessaire pour son déroulement correct.

5.2.3.1 Prologue du traitement de la signification.

Comme nous l'avons fait pour les paramètres, nous allons utiliser des fonctions du compilateur afin de traiter la signification. Pour ce faire nous avons défini une fonction qui sert à contrôler ce processus. Les actions qu'elle acomplit sont:

- Etablir dans le système l'état "traitement de la signification".
- Générer un premier composant du programme abstrait, constitué par l'appel d'une fonction spéciale. Lors de l'interprétation cette fonction sauvera l'environnement de travail afin de le restituer lorsqu'on revient au langage de base.
- Appel du compilateur dans la routine

correspondant à la classe syntaxique qu'on étend.

Le comportement du compilateur lors de l'état "traitement de la signification" implique des modifications vis-à-vis de son déroulement normal. Ces modifications concernent l'interpréteur de l'analyseur syntaxique, et elles servent à l'analyse syntaxique de la signification, à la construction du programme abstrait et à la sauvegarde de l'environnement de travail.

5.2.3.2 Analyse syntaxique de la signification.

D'une 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 des terminaux 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.

Pour l'analyse de la signification on trouve qu'elle comporte également des éléments non-terminaux qui correspondent aux paramètres formels. Donc, dans ce cas le processus d'analyse peut se représenter par un arbre dont certaines feuilles restent à élaborer.

Afin de réaliser ce processus on a dû modifier les critères d'analyse qu'utilise le compilateur. Rappelons que ces critères se basent sur le fait que la grammaire du langage de base est LL(1). L'algorithme est déterministe et utilise l'entité lexicographique courante pour diriger son déroulement. Le processus modifié d'analyse consiste à parcourir l'arbre syntaxique jusqu'à trouver la routine associée à la classe syntaxique du paramètre. A ce moment on simule un "RETURN" comme si la routine avait été exécutée, et on continue l'analyse normale. Le problème qui reste à résoudre consiste à déterminer la façon de réaliser correctement ce parcours. Dans l'analyseur initial on utilise l'entité lexicographique courante qui indique le chemin dans l'arbre. Pour le système modifié nous allons utiliser les éléments "premiers" des classes syntaxiques. Dans cette perspective l'interpréteur évaluera les pseudo-instructions DECIDE d'une manière différente. Au lieu de comparer le paramètre du DECIDE avec l'entité lexicographique courante, il le fera avec tous les éléments "premiers" de la classe syntaxique du paramètre. En cas de succès d'une comparaison on continue l'interprétation selon le processus d'origine, et ceci jusqu'à trouver un appel de la routine associée à la classe syntaxique.

Voyons un exemple explicatif de ce processus d'analyse. Prenons la grammaire:

```
X:= x A
A:= B | a
B:= M D | b
D:= d
M:= m
```

Où X est l'axiome. L'ensemble des "premiers" est:

```
X : (x)
A : (m,b | a)
B : (m | b)
M : (m)
D : (d)
```

L'analyseur, sous la forme de macro-instructions, contient, entre autres, les routines suivantes:

```
ROUTINE  X
CHECK (x)
ENTER  A
RETURN
```

```
    ROUTINE  A
    DECIDE,L1,SINGLE,b
             SINGLE,m
    CHECK (a)
    EXIT
L1  ENTER  B
    RETURN

    ROUTINE  B
    DECIDE,L2,SINGLE,m
    CHECK (b)
    EXIT
L2  CALL  M
    ENTER  D
    RETURN
```

Supposons qu'on désire analyser syntaxiquement la chaîne non-terminale:

x M

Après la vérification de l'ocurrence de "x", le pré-processeur avance dans le texte source et il trouve le nom de la classe syntaxique "M". L'analyseur commence à exécuter la routine "A" et il arrive au DECIDE. La première comparaison est avec "b", et il la réalise avec les "premiers" de M. Comme l'essai est infructueux l'analyseur

réalise la deuxième comparaison qui dans ce cas réussit. Donc il se branche vers L1. Lors de l'exécution de la routine "B" il procède d'une manière analogue, et il trouve l'appel à "M". A ce moment un appel est fait au pré-processeur; l'analyseur exécute ensuite l'instruction suivante sans entrer dans la routine "M".

Il peut paraître surprenant qu'avec cette modification de l'analyse nous n'ayons pas tenu compte des éléments "successeurs" des classes syntaxiques qui génèrent le mot-vide, car ceci est considéré par l'algorithme initial. La raison de cette absence vient du fait que le Transformateur de Grammaires, lorsqu'il produit la routine associée à une classe syntaxique, génère autant de comparaisons (DECIDE) qu'il existe d'alternatives de la règle de productions moins une. La séquence dans la routine après les DECIDE correspond à l'alternative non considérée. Ainsi dans la grammaire de l'exemple précédent, la classe syntaxique "B" comporte deux alternatives de génération. Dans la routine qui lui est associée, on trouve un seul DECIDE qui teste une de ces alternatives et la séquence correspond implicitement à l'autre. Dans la version du Transformateur de Grammaires dont nous disposons, lorsqu'une classe syntaxique génère le mot-vide, les comparaisons dans la routine associée se font avec les "premiers", et le reste correspond à la génération du mot-vide. Ceci veut dire qu'il n'y a pas d'utilisation des "successeurs" dans les DECIDE,

et donc, pour notre modification nous n'en tiendrons pas compte.

Nous ne présenterons pas ici une démonstration formelle du fait que la méthode d'analyse modifiée conduit à l'acceptation des phrases non-terminales du langage étendu. D'une manière intuitive le fait de travailler avec les "premiers" constitue un cas plus général que l'utilisation de l'entité lexicographique; donc d'une certaine manière ces modifications signifient faire le chemin inverse de celui réalisé pour construire l'analyseur, ce qui assure la validité des modifications.

5.2.3.3 Construction du programme abstrait.

La construction du programme abstrait de la signification implique deux types d'actions: capture et sauvegarde des appels aux fonctions sémantiques avec leurs environnements associés, et établissement des liens pour les paramètres.

La première action est réalisée par l'interpréteur, qui dans l'état "traitement de la signification" et lors des appels aux fonctions sémantiques, sauvegarde cette information dans la zone allouée pour le programme abstrait. Nous avons déjà signalé que pour l'interprétation du programme abstrait il ne suffit pas d'avoir la suite des

fonctions sémantiques à évaluer, mais également leur environnement de travail.

La construction du programme abstrait implique ainsi une recopie des appels aux fonctions sémantiques qu'on détecte pendant l'analyse de la signification. Pour tenir compte des environnements de travail une nouvelle pseudo-instruction a été créée dont la syntaxe est la suivante:

code nom-fct-sém ptr-env

où:

code est la nouvelle pseudo-opération (numéro 17).

nom-fct-sém est le nom (numéro) de la fonction sémantique.

ptr-env est un pointeur vers l'environnement correspondant.

La sémantique de cette pseudo-instruction correspond à la restitution des variables globales suivi de l'appel à la fonction.

Pour les paramètres, et après qu'on ait trouvé la routine associée selon le processus décrit dans le paragraphe précédent, on réalise une recherche dans la zone où ils sont alloués. Il faut rappeler que lors du traitement

de la structure, on avait alloué deux emplacements pour chaque paramètre: un pour le nom de la routine, et l'autre pour le numéro indicatif. La recherche consiste à associer le paramètre qui apparaît dans la signification avec cette information. Pour ce faire nous utilisons la pseudo-instruction d'appel à une fonction syntaxique. Ceci assure le retour au programme abstrait de la signification après l'interprétation de celui du paramètre, comme nous le verrons par la suite.

Le programme abstrait ainsi construit est essentiellement une suite d'appels aux fonctions sémantiques comportant éventuellement des appels syntaxiques utilisés pour les paramètres. Dans le cas où les paramètres réels comportent de nouvelles utilisations d'extension, cette structure devient arborescente.

L'ordre des fonctions sémantiques dans le programme abstrait a été obtenu lors de l'analyse syntaxique de la signification. La validité du processus d'interprétation découle du respect de cet ordre.

5.2.3.4 Sauvegarde des environnements de travail.

Le compilateur GSL utilise une zone de variables globales permettant de communiquer entre les différentes fonctions. Parmi ces informations on trouve celles fournies

par le pré-processeur lors de chaque lecture du texte source, et, notamment le caractère lexicographique courant. Mais, comme nous l'avons déjà dit, l'analyse des identificateurs, des constantes arithmétiques et des constantes de caractères, est faite au niveau du pré-processeur. Donc il ne suffit pas de donner l'entité lexicographique quand il s'agit d'un de ces éléments. Ainsi pour les identificateurs il faut connaître leur entrée dans la table correspondante ainsi que leur nom; pour les constantes nous devons connaître leur valeur. Tous ces renseignements sont mis dans des variables globales qui constituent l'information qu'on doit sauvegarder pour l'exécution correcte du programme abstrait.

Dès que l'état "traitement de la signification" est établi, on introduit un interface entre le système et le pré-processeur de sorte qu'à chaque activation de ce dernier, l'information qu'il fournit soit sauvegardée, et la valeur des variables globales qui pointent vers ces données soit mise à jour. Ce sont les valeurs de ces variables qu'on utilise pour le paramètre "ptr-env" des nouvelles pseudo-instructions composant le programme abstrait et que nous avons présenté dans le paragraphe précédent.

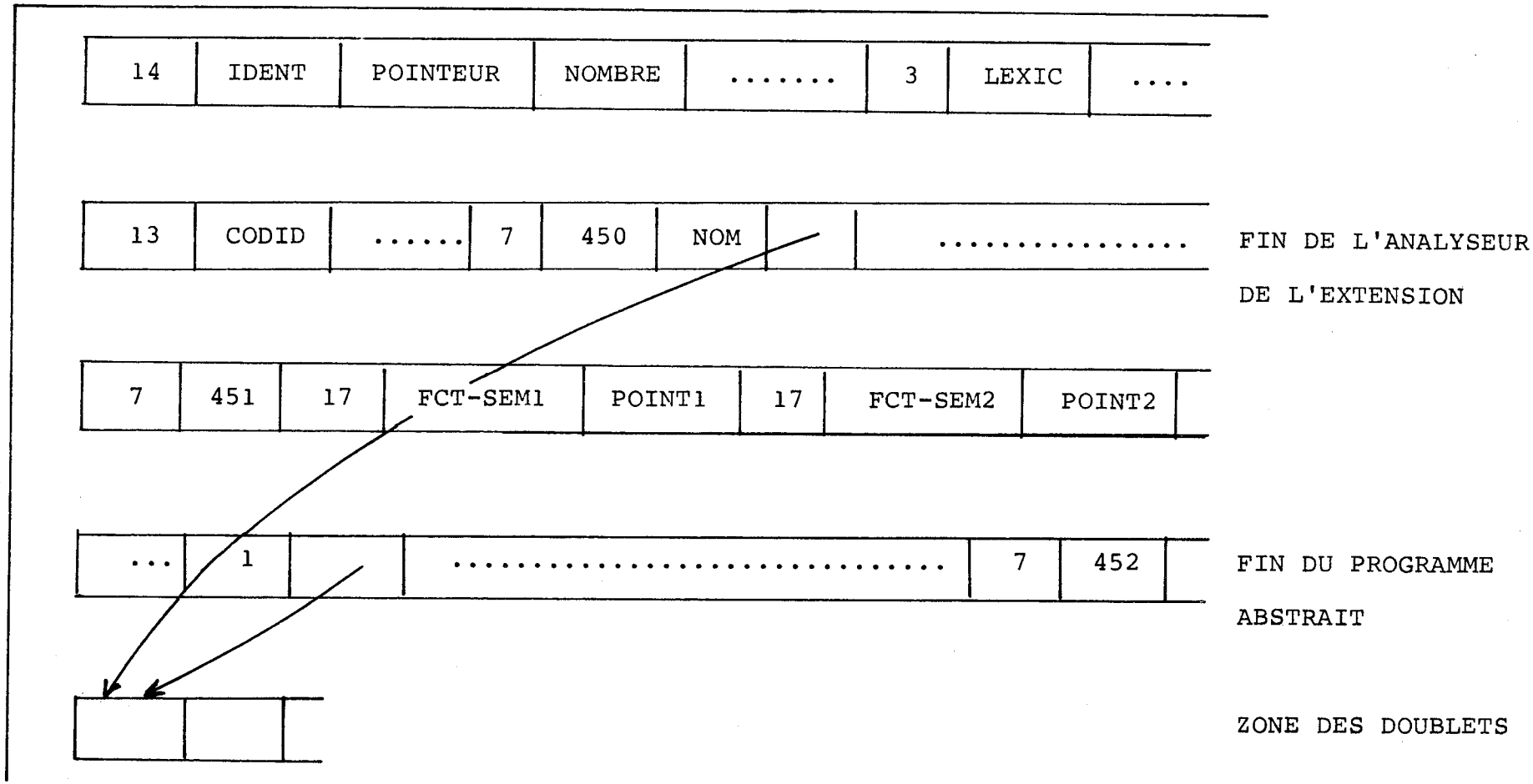
5.2.3.5 Epilogue du traitement de la signification.

Le traitement de la signification a été réalisé par des routines du compilateur, à partir de l'appel effectué par la fonction sémantique décrite en 5.2.3.1. Lorsque ce processus est terminé on utilise une nouvelle fonction chargée de restituer les conditions normales pour le déroulement du compilateur.

Les actions qu'elle doit accomplir sont:

- Fermeture du programme abstrait avec l'incorporation de l'appel à une fonction qui restitue l'environnement de travail original.
- Sortie de l'état "traitement de la signification".

De cette façon s'achève le traitement de la définition d'une extension. Le compilateur a été dynamiquement modifié de manière à reconnaître et traiter le nouveau langage étendu. Dans la figure 5.3 nous avons schématisé l'ensemble: analyseur d'une utilisation et programme abstrait.



PSEUDO-OPERATEURS

1 appel à une fonction syntaxique
3 check
7 appel à une fonction sémantique
13 test d'un identificateur
14 entête de l'analyseur
17 appel à une fonction sémantique dans le programme abstrait

FONCTIONS

450 constructeur des programmes abstraits des paramètres
451 sauvegarde des environnements de travail
452 restitution des environnements de travail

FIGURE 5.3

5.3 Interprétation du langage étendu.

L'interpréteur du compilateur doit être modifié afin de pouvoir reconnaître les nouvelles pseudo-instructions qu'on a créées pour le traitement des extensions. Dans la conception du système nous nous sommes efforcés d' utiliser le plus souvent possible le compilateur existant. Ainsi nous avons défini seulement trois nouveaux codes interprétables, à savoir:

- En-tête de l'analyseur des extensions, afin de vérifier le premier élément de la structure.
- Vérification d'un identificateur particulier de la structure.
- Appels à des fonctions sémantiques dans les programmes abstraits, afin d'inclure comme paramètre l'adresse de leurs environnements de travail.

L'existence des deux premières pseudo-instructions vient du fait que le traitement des identificateurs est réalisé au niveau du pré-processeur. Dans une autre conception du compilateur du langage de base on peut envisager de modifier dynamiquement la table du pré-processeur de façon à donner aux identificateurs de la structure la caractéristique de mot-clé, et leur affecter une entité lexicographique. Dans cette perspective, les deux

premières pseudo-instructions sont superflues, et il suffit de travailler avec l'interpréteur initial.

5.3.1 Analyse d'une utilisation.

Le traitement d'une utilisation est réalisé par les différentes fonctions présentées dans les paragraphes précédents. L'analyse syntaxique sera faite par l'analyseur des extensions, et activée si et seulement si, l'utilisation se trouve dans le contexte de la classe syntaxique étendue. Pour les paramètres réels on construira leurs programmes abstraits, sous le contrôle d'une fonction ad-hoc dont l'appel a été incorporé dans l'analyseur de l'extension. Ce processus correspond à l'appel d'une routine du compilateur comme ce fut le cas pour la routine de traitement de la signification d'une définition. Avant de procéder à cet appel, il faut réserver de la place pour le programme abstrait du paramètre. Rappelons que dans la zone de données qui correspond à la chaîne codée de l'analyseur, nous avons prévu un emplacement destiné à recevoir les programmes abstraits des paramètres. Le travail dans cette région est fait par l'intermédiaire d'une pile qui assure les utilisations imbriquées des extensions. Lorsqu'on commence l'analyse syntaxique d'un paramètre réel, on initialise la zone des paramètres avec cette pile ainsi que la zone de son programme abstrait. Pour ce faire on procède de la façon

suivante:

i) Un des paramètres de la pseudo-instruction d'en-tête de l'analyseur des extensions, contient le numéro des paramètres. En utilisant cette information on connaît le nombre de doublets associé à l'extension. Donc on réserve la zone pour tous les doublets, on laisse ensuite l'espace pour les programmes abstraits.

ii) Le second paramètre d'appel de la fonction qui contrôle la construction du programme abstrait du paramètre, contient l'adresse du doublet qui lui est associé. Dans ce doublet on incorpore un branchement vers la zone libre obtenue en i).

iii) On construit le programme abstrait dans la zone appropriée.

iv) On met à jour l'adresse relative de la région libre afin de pouvoir l'utiliser pour d'autres paramètres de l'extension.

5.3.2 Evaluation de l'utilisation.

Lorsque l'analyse de l'utilisation est terminée, on interprète le programme abstrait. Le processus consiste à parcourir l'arbre abstrait final qui représente l'extension

en évaluant les différents appels des fonctions sémantiques avec les environnements associés. Pour les paramètres, les appels qu'on inclut dans le programme abstrait de la signification, vers leurs doublets correspondants, nous assurent:

i) L'accès au programme abstrait du paramètre, au moyen du branchement placé dans le doublet.
ii) Le retour au composant suivant dans le programme de la signification, à la fin de l'évaluation du paramètre.

A la suite de cette évaluation on manipule la pile associée à la zone des paramètres de façon à libérer la region qu'on vient d'utiliser.

La configuration trouvée par l'interpréteur est présentée dans la figure 5.4.

Avec l'interprétation du programme abstrait on achève le processus de manipulation des utilisations. Il est possible maintenant de mettre en évidence les deux "passages" que comporte l'évaluation des extensions:

i) Pendant le traitement de la définition, analyse de la signification et construction du

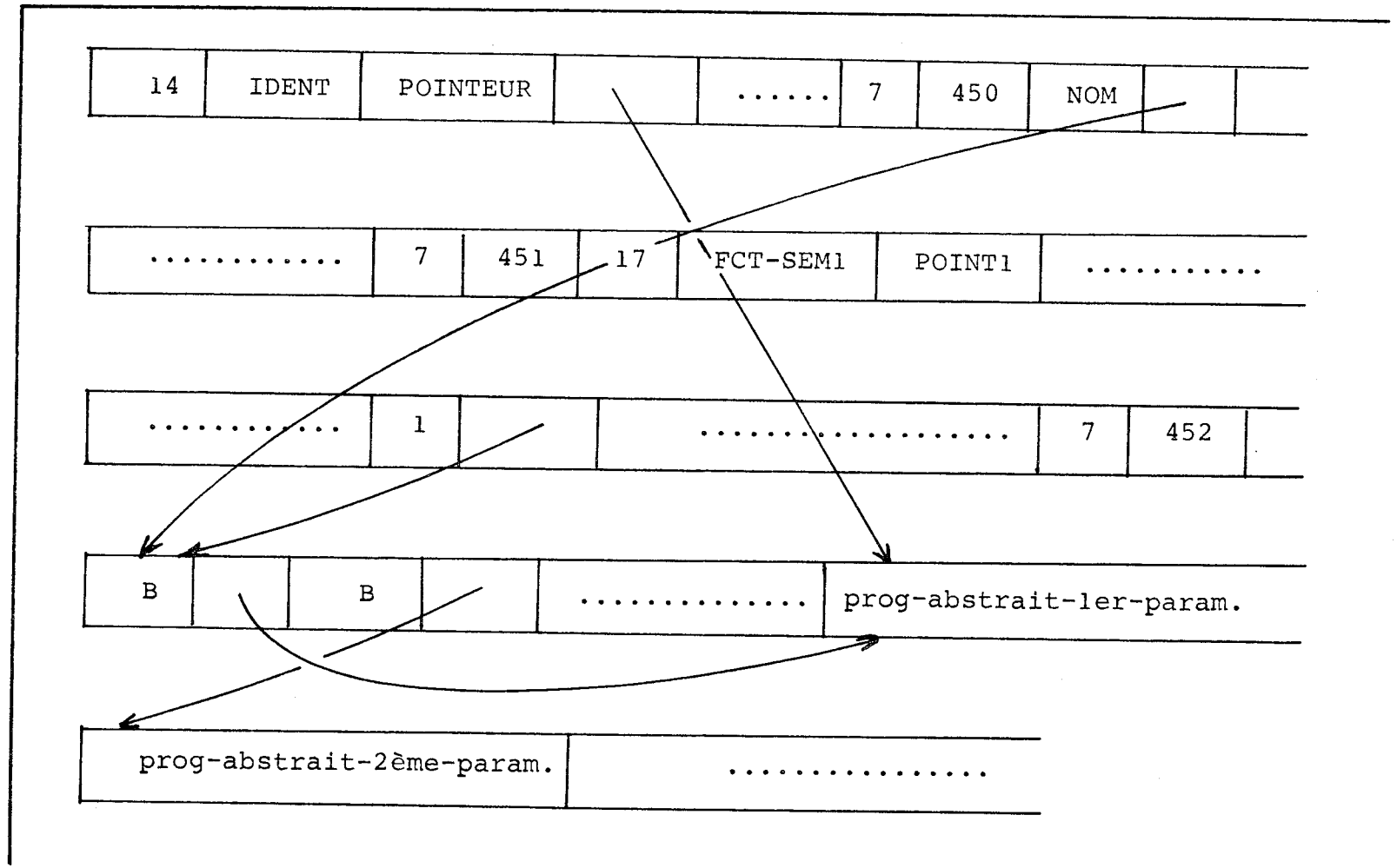

14
IDENT
POINTEUR
.....
7
450
NOM
.....
7
451
17
FCT-SEM1
POINT1
.....
1
.....
7
452
B
B
.....
prog-abstrait-1er-param.
prog-abstrait-2ème-param.
.....

FIGURE 5.4

programme abstrait partiel correspondant

ceci, fait une seule fois, reste valable pour toutes les utilisations.

Pendant le traitement des utilisations on complète le programme abstrait de la signification par les programmes abstraits des paramètres réels, afin d'obtenir le programme abstrait final de l'extension.

ii) L'interpréteur traite ce programme abstrait dans un deuxième passage qui complète l'évaluation de l'extension.

5.4 Conclusion.

Nous avons obtenu avec cette réalisation l'implémentation du système dont la présentation est faite dans le chapitre 3.

Le comportement du compilateur GSL a permis l'insertion aisée des fonctions du système, et le travail de programmation pour ce cas particulier ne fut pas énorme vis à vis des résultats. De plus nous disposons maintenant pour l'application à un autre langage d'outils de départ tel que le schéma des modifications du compilateur, les fonctions sémantiques, et le nouvel interpréteur.

Du point de vue de l'utilisation du système on a pu constater que l'efficacité a priori de la méthode d'évaluation est atteinte dans la réalisation, et les temps d'exécution sont satisfaisants.

C O N C L U S I O N .

Ce travail présente un système d'extension syntaxique pour les langages de programmation. Le concept d'extension syntaxique a été souvent utilisé dans la littérature; nous en avons donné une définition précise dans le chapitre 1.

Le rappel des différentes propositions d'extension syntaxique fait dans le chapitre 2, a servi à mettre en évidence les difficultés rencontrées pour déboucher sur des systèmes éfficaces. Une proposition essayant d'apporter une solution à ces problèmes, est présentée dans le chapitre 3. Ses caractéristiques principales sont:

- Conception du système sur un modèle de compilateur largement utilisé (Méthode LL(1)).

- Evaluation des extensions en réalisant une compilation partielle des définitions d'extension, et en travaillant avec leurs programmes abstraits.

Avec la première condition on libère le système d'un langage particulier. Le fait que le modèle de compilateur utilise une méthode d'analyse syntaxique qui soit déterministe et descendante permet d'espérer un champ assez

grand d'applications, car cette méthode est actuellement très utilisée pour la construction de compilateurs.

La deuxième condition offre les avantages suivants:

- d'une part on réalise au moment de la définition d'une extension une analyse syntaxique de la signification, et on avance dans la compilation le plus possible.

- d'autre part ce processus sert à réaliser un ensemble de vérifications qui permettent de détecter les erreurs, au moment du traitement de la définition.

Sur la base d'une telle conception nous avons réalisé une implémentation particulière, de façon à vérifier les objectifs de départ, et afin qu'elle puisse être utilisée comme modèle pour l'application à d'autres langages. Cette réalisation a été faite en deux parties:

- une première, indépendante de tout langage, avait comme objectif l'obtention d'un compilateur qui ayant les caractéristiques du modèle, permette l'insertion aisée du système d'extension (chapitre 4).

- une deuxième partie a été constituée par l'application à un langage particulier (chapitre 5). Les résultats obtenus ont demontré que les objectifs de départ ont été atteints.

En conclusion de cette expérience on peut faire les remarques suivantes:

- La méthode de Vienne, méthode formelle de définition de langages, peut être utilisée partiellement comme schéma du processus de compilation, et que les concepts utilisés correspondent à des fonctions du compilateur réel. La distinction entre définition et implémentation des langages, peut ainsi être atténuée en utilisant cette méthode pour la définition du langage.

- Le système proposé ici est un moyen souple et efficace de définir un langage étendu. La souplesse est obtenue par l'absence de classes syntaxiques privilegiées pour les extensions. L'efficacité est atteinte par la méthode d'évaluation des extensions. En ce qui concerne le principe d'utilisation du système on reste dans le domaine de la description syntaxique des langages, qui est un sujet bien connu. Mais l'obtention de toute la puissance du

système nécessite une connaissance approfondie du langage de base, ce qui n'est évidement pas à la portée de tous les utilisateurs. La création d'un langage étendu pour un groupe d'utilisateurs doit donc être fait par un expert, qui alliant sa connaissance des besoins du groupe à sa compétence dans le langage de base, peut tirer pleinement partie de la souplesse et de l'efficacité du système.

- Les possibilités offertes par un compilateur admettant des extensions ont été décrites par Schuman et Jorrand (31). Nous apporterons les précisions suivantes:

  - Dans la plupart des langages de haut niveau le concept de procédure permet d'écrire des algorithmes. Dans un environnement d'extension syntaxique l'utilisateur a le choix de la classe syntaxique et il dispose de la vérification syntaxique des paramètres.

  - Parmi les récents efforts pour réunir les concepts de définition et d'implémentation des langages de programmation, on trouve la thèse de Willis (38). A partir d'un noyau de primitives, la définition d'un langage est construite comme une extension de ce noyau. Le fait de disposer d'un compilateur pour le noyau, et d'un système d'extension assure l'implémentation du nouveau

langage. Un avantage supplémentaire réside dans le fait que la "portabilité" du nouveau langage devient facile du moment que la partie à transporter se réduit au compilateur du noyau.

- La réalisation décrite au chapitre 5 a permis de constater la validité du système. Une partie du travail réalisé reste valable pour l'application à d'autres compilateurs. Ce travail est essentiellement constitué par les modifications au Transformateur de Grammaires, afin d'obtenir des sorties partielles et des informations suplémentaires. Ces modifications et les schémas des routines du système conçus comme des fonctions sémantiques, permettent d'envisager la réalisation d'un système analogue en un temps relativement court.

Ce travail est du domaine des extensions syntaxiques. Bien que nous ayons aussi présenté et défini dans la chapitre 1, les extension sémantiques. On dispose à l'heure actuelle, de systèmes qui permettent de les utiliser. Une suite logique du travail serait de définir un système qui comporte aussi bien des extensions syntaxiques, que des extensions sémantiques. Par exemple, Bert (2) a réalisé une étude des éléments fondamentaux des langages de programmation, en ce qui concerne le contrôle de

l'utilisation des objets et des primitives d'exécution. Ceci l'a amené à définir un langage où il compose ces éléments au moyen d'extensions sémantiques. L'application à ce langage de notre proposition d'extension permettrait de lui ajouter des contrôles syntaxiques, complétant ainsi un ensemble de vérifications qu'il est souhaitable d'avoir dans les langages de programmation.

Une autre voie possible de recherche, est d'essayer de cerner l'influence de tels systèmes sur la méthodologie de la programmation. Le souci croissant du développement de l'informatique est de "mieux programmer". Pour cela on a envisagé des méthodologies, par exemple la programmation structurée (9). On ne trouve que de rares efforts destinés à automatiser ces méthodes. Un système d'extension syntaxique pourra être la base d'une telle réalisation dans la mesure où il permet de construire les différents niveaux d'une programmation par raffinements succesifs (40).

A N N E X E I

PSEUDO-INSTRUCTIONS DE LA CHAINE INTERPRETABLE.

Pendant le procesus d'interprétation on dispose d'un pointeur de contrôle pointant vers la routine en cours d'évaluation.

1) Appel d'une routine syntaxique.

syntaxe: 1 nom-rout

où: nom-rout correspond à une entrée dans la chaîne codée où commence la routine syntaxique appelée.

sémantique: L'interpréteur sauvegarde l'adresse de la pseudo-instruction suivante dans la pile, et met à jour le pointeur de contrôle avec nom-rout.

macro-opération correspondante: CALL

2) Entrée dans une routine syntaxique.

syntaxe: 2 nom-rout

où: nom-rout correspond à une entrée dans la chaîne codée où commence la routine syntaxique appelée.

sémantique: L'interpréteur met à jour le pointeur de contrôle avec nom-rout.

macro-opération correspondante: ENTER

3) Test d'une entité lexicographique.

syntaxe: 3 ent-lex

où: ent-lex représente une entité lexicographique.

sémantique: On compare ent-lex avec l'entité lexicographique courante. En cas d'égalité on appel le pré-processeur afin d'avancer dans le texte source. Sinon on appel la fonction de traitement des erreurs.

macro-opération correspondante: CHECK

4) Décision sur les symboles directeurs.

syntaxe: 4 etiq n elt-lex-1 ent-lex-2

où: etiq est un pointeur vers une entrée dans la chaîne codée.

n est un entier qui peut prendre les valeurs 1 ou 2.

ent-lex-1 et ent-lex-2 représentent des entités lexicographiques.

sémantique: Si n=1 le paramètre ent-lex-2 n'existe pas. Dans ce cas on compare ent-lex-1 avec l'entité lexicographique courante. En cas d'égalité on met à jour le pointeur de contrôle avec la valeur de etiq. Sinon ce ponteur de contrôle prend la valeur de l'entrée de l'instruction suivante dans la chaîne.

Si n=2 on test si l'entité lexicographique courante appartient à l'intervalle définit par ent-lex-1 et ent-lex-2. En cas de réussite on met à jour le pointeur de contrôle avec la valeur de etiq. Sinon ce pointeur de contrôle prend la valeur de l'entrée de l'instruction suivante dans la chaîne.

macro-opération correspondante: DECIDE

5) Retour d'une routine.

syntaxe: 6

sémantique: On met à jour le pointeur de contrôle avec la valeur de tête de la pile.

macro-opérations correspondantes: RETURN , EXIT

6) Appel d'une fonction sémantique.

syntaxe: 7 nom-fct-sém

où: nom-fct-sém représente le nom de la fonction sémantique utilisé par le Transformateur de Grammaires.

sémantique: L'interpréteur commence par traduire nom-fct-sém dans le nom de la fonction appelé. Pour ce faire il utilise les liens établis par l'editeur de liens du système d'exploitation. L'évaluation de

l'instruction consiste à sauvegarder l'adresse de la routine suivante dans la chaîne, et donner le contrôle à la routine appelée.

macro-opération correspondante: CALL

7) Entrée dans une fonction sémantique.

syntaxe: 8 nom-fct-sém

où: nom-fct-sém représente le nom de la fonction sémantique utilisé par le Transformateur de Grammaires.

sémantique: La sémantique est analogue à celle de la pseudo-instrucion d'appel d'une fonction sémantique sauf qu'on ne sauvegarde pas l'adresse de l'instruction suivante.

macro-opération correspondante: ENTER

8) Branchement dans la chaîne.

syntaxe: 9 nom-fct

où: nom-fct correspond à une entrée dans la chaîne codée où commence la routine syntaxique.

sémantique: L'interpréteur met à jour le pointeur de contrôle avec la valeur de nom-fct.

opération correspondante: B

Schéma de l'interpréteur.

Nous avons choisi le langage ALGOL 68 pour décrire l'interpréteur. A chaque pseudo-instruction correspond une déclaration de mode; l'interpréteur utilise ce mode pour déterminer les actions à entreprendre.

```
MODE call-syntaxique= INT;
MODE return= VOID;
MODE enter-syntaxique= INT;
MODE check= INT;
MODE call-semantique= REF PROC;
MODE enter-semantique= REF PROC;
MODE decide= FLEX[1:0]INT;
MODE pseudo-instruction= UNION(CALL-SYNTAXIQUE,ENTER-SYNTAXIQUE,
                               RETURN,CHECK,CALL-SEMANTIQUE,
                               ENTER-SEMANTIQUE,DECIDE);
[1:5000]PSEUDO-INSTRUCTION chaine-codee;
INT index: =1;
[1:200]INT pile;
INT indexpile:=0;
PROC empile = (INT i) (indexpile:=indexpile+1;pile[indexpile]:=i);
PROC depile = INT (indexpile:=indexpile-1; pile[indexpile+1] );


WHILE index <= UPB chaine-codee DO
  CASE chaine-codee[index] IN
    (CALL-SYNTAXIQUE x): (empile(index); index:=x);
    (ENTER-SYNTAXIQUE x):  index:=x;
    (CHECK x):  IF currchar=x THEN preproc ELSE error FI;
    (RETURN x):  index:=depile ;
    (CALL-SEMANTIQUE x): (x; index:=index + 1);
    (ENTER-SEMANTIQUE x): (x; index:=depile);
    (DECIDE x): (IF x[2]=1
                  THEN IF x[3]=currchar
                          THEN index:=x[1]
                          ELSE index:=index + 1
                       FI
                  ELSE IF currchar >= x[3]
                          THEN IF currchar <= x[4]
                                  THEN index:=x[1]
                                  ELSE index:=index + 1
                               FI
                          ELSE index:=index + 1
                       FI
                FI)
  ESAC
OD;
```

A N N E X E I I

EXEMPLE D'UTILISATION DU SYSTEME.

```
F:PROC;
EXTSYN <STA>:= RECV MAVC |->    A=B+YL FINEXT;
EXTSYN <RESTCFEX>:= VCYCNS |-> AA-UU FINEXT;
EXTSYN <FLEM>:= MAIN |->  (RR-ZZ) FINEXT;
EXTSYN <STA> := EXEMP |-> XX=MAIN-(VCYCNS) FINEXT;
DC;
RECV MAVC;
GH=JK-MAIN;
EXEMP;
END;
EXTSYN <STA>:= ERRE1 |-> IF AA>5 TIEN A=8 FINEXT;
                                          *
          ERROR  007
EXTSYN <RESTCFEX> := ERRE2 ERRE3 ERR4 |-> VB-JK FINEXT;
EXTSYN <NOM> := NOM DE FAMILLE DE JEAN |-> JEAN FINEXT;
AA=REC-NOM DE FAMILLE DE JEAN;
GS=TEG-4*VB// ERRE2 ERRE3 ERRE5;
                                          *
          ERROR  007
END;
```

007 S-SYNTAX ERROR

```
FILE: EXEMPLE  ASSEMELE A  GRENOBLE SCIENTIFIC CENTRE

*EXEMP;
          L       9,FR
          S       9,ZZ
          L       8,AA
          S       8,UU
          SR      9,8
          ST      9,XX
*END;
*EXTSYN <STA>:= ERRE1 |->  IF AA>5 THEN A=8 FINEXT;
*EXTSYN <RESTOFEX> := ERRE2 ERRE3 ERR4 |-> VB-JK FINEXT;
*EXTSYN <NOM> := NOM DE FAMILLE DE JEAN |-> JEAN FINEXT;
*AA=RED-NOM DE FAMILLE DE JEAN;
          L       8,RED
          S       8,JEAN
          ST      8,AA
*CS=TFG-4*VB// ERRE2 ERRE3 ERRE5;
          L       7,TFG
          S       7,=C'C'
          M       6,VB
*END;
          BR      8
=C'C'     DC      F'CCCCCCCC'
8         CSECT
          DS      0D
=DATA     EQU     *
A         EQU     =DATA+00000000          DS    F
P         EQU     =DATA+00000004          DS    F
YI        EQU     =DATA+00000008          DS    F
GH        EQU     =DATA+0000000C          DS    F
JK        EQU     =DATA+00000010          DS    F
RF        EQU     =DATA+00000020          DS    F
ZZ        EQU     =DATA+00000024          DS    F
XX        EQU     =DATA+00000028          DS    F
AA        EQU     =DATA+0000002C          DS    F
LI        EQU     =DATA+00000030          DS    F
RED       EQU     =DATA+00000040          DS    F
JEAN      EQU     =DATA+00000044          DS    F
CS        EQU     =DATA+00000048          DS    F
TFG       EQU     =DATA+00000052          DS    F
VB        EQU     =DATA+00000056          DS    F
          ORG     =DATA
          DS      0L60
          END
```

```
          TITLE 'FILE EXEMPLE '
*P:PROC;
          PRINT OFF
          MACRO
          CLP &N,&S,&T
          LCLA  &A
&A SETA 0
          AIF (K'&T EQ 0).C
.DD AIF (&A EQ &N).E
          CC &S,&T
&A SETA &A+1
          AGO .DD
.C CC &N&S
.E MEND
          MEND
          PRINT ON,NOGEN
P         CSECT
          PRINT OFF
ð0        EQU   00
ð1        EQU   01
ð2        EQU   02
ð3        EQU   03
ð4        EQU   04
ð5        EQU   05
ð6        EQU   06
ð7        EQU   07
ð8        EQU   08
ð9        EQU   09
ðA        EQU   10
ðB        EQU   11
ðC        EQU   12
ðD        EQU   13
ðE        EQU   14
ðF        EQU   15
          PRINT ON,NOGEN
          BALR  ðE,0
          USING *,ðB
ðBP       EQU   *
          USING ðP,ðC
          L     ðC,=A(ðP)
*EXTSYM <STA>:= RECV MOVE |->   A=B+YU FINEXT;
*EXTSYM <RESTOFEX>:= VOYONS |-> AA-LL FINEXT;
*EXTSYM <FLEM>:= MAIN |->  (RR-ZZ) FINEXT;
*EXTSYM <STA> := EXEMP |-> XX=MAIN-(VOYONS) FINEXT;
*DO;
*RECV MOVE;
          L     ðD,B
          A     ðD,YU
          ST    ðD,A
*GH=JK-MAIN;
          L     ðA,RR
          S     ðA,ZZ
          LCR   ðA,ðA
          A     ðA,JK
          ST    ðA,GH
```

B I B L I O G R P H I E .

(1) ASSABGUI M.

Interprétation d'une chaîne codée générée par un transformateur de grammaires.
Université S. et M. de Grenoble. Mai 1969.

(2) BERT D.

Etude d'éléments fondamentaux des langages de programmation: contrôle de l'utilisation des objets et primitives d'exécution.
Thèse de 3ème. Cycle. Mai 1973
Université S. et M. de Grenoble.

(3) BERTHAUD M.

Note interne.

(4) BERTHAUD M., CLAUZEL D., JACOLIN M.

GSL (Grenoble System Language) Définition du langage.
Centre Scientifique IBM, Grenoble.
Etude FF2-0133, Janvier 1972.

(5) BROWN P.J.

A survey of macro processors.
Automatic Programming. Vol. 6
Pergamon Press.

(6) BROWN P.J.

The ML/I macro processor.
Communications of A.C.M. Oct. 1967

(7) CHATELIN Ph., WILLIS B.

Le mécanisme des macros syntaxiques.
Rapport scientifique. Contrat CRI 70-107
Decembre 1972.

(8) CHEATHAM T.E.

The introduction of definitional facilities into higher level programming languages.
Proc. Fall Joint Comp. Conf.
Vol 29. Novembre 1966.

(9) DAHL O.J., DIJKSTRA E.W., HOARE C.A.R.

Structured Programming.
Academic Press. 1972.

(10) DIJKSTRA E.W.

On the necessity of correctness proofs.
Advanced Course on Computer Systems Architecture.
Alpe d'Huez. Decembre 1972.

(11) EARLEY J.

An efficient context free parsing algorithm.
Communications of A.C.M. Feb. 1970

(12) FOSTER J.M.

A syntax improving device.
Computer Journal. Mai 1968.

(13) GALLER B.A., PERLIS A.J.

A proposal for definitions in ALGOL.
Communication of A.C.M. Avril 1967.

(14) GRIFFITHS M., PELTIER M.

Grammar transformation as an aid to compiler production.
Centre Scientifique IBM, Grenoble.
Etude FF2.0057.0 . Mai 1968.

(15) GRIFFITHS M.

Relationship between definition and implementation.
Advanced course in software engineering.
Lecture Notes in Economics and Mathematical Systems, 81.
Spring Verlag, 1973.

(16) GRIFFITHS M.

LL(1) Grammars and Analysers.
Advanced Course on Compiler Construction.
Technical University of Munich.
Munich. Mars 1974.

(17) HOARE C.A.R., WIRTH N.

An axiomatic Definition of the Programming Language PASCAL.
International Summer School on Structured Programming and Programmed Structures.
Munich 1973.

(18) JACQUET P., VIDART J.

Compilateur dirigé par l'interprétation de l'analyseur syntaxique.
Note Interne.
Janvier 1974.

(19) JORRAND Ph.

Some aspects of BASEL, the base language for an extensible langguage facility.
Extensible Language Symposium. Boston.
SIGPLAN Vol 4, Nro. 8. Août 1969.

(20) JORRAND Ph., BERT D.

On some basic concepts for extensible programming languages.
International Computing Symposium.
Venise. Avril 1972.

(21) KNUTH D.E.

Top down syntax analysis.
International Summer School on Computer Programming.
Copenhague. Août 1967.

(22) KOSTER C.H.A.

A compiler compiler.
Mathematisch Centrum. Amsterdam.
MR 127. 1971.

(23) LEAVENWORTH B.M.

Syntax macros and extended translation.
Communications of A.C.M. Novembre 1966.

(24) LUCAS P., LAUER P., STIGLEITNER H.

Method and notation for the formal definition of programming languages.
IBM Laboratory of Vienna.
TR 25.087. 1968.

(25) MANDIL S.

A brief description of MP/3.
IBM. United Kingdom.

(26) McCARTHY J.

LISP 1.5 programmer's manual.
The MIT Press 1962.

(27) McCARTHY J.

Towards a mathematical science of computation.
Information Processing 1962.
North Holland Publ. Comp. Amsterdam 1963.

(28) McILROY M.D.

Macro instructions extensions of compiler languages.
Communications of A.C.M. Avril 1960.

(29) MOERS C.N., DEUTCH L.P.

TRAC, a procedure describing languages for the reactive typewriter.
Communications of A.C.M. Mars 1966.

(30) PELTIER M.

The macro-system.
Centre Scientifique IBM, Grenoble.
Etude FF2.0057. Mai 1968.

(31) SCHUMAN S., JORRAND Ph.

Definition mechanisms in extensible programming languages.
Fall Joint Computer Conference. 1970.
AFIPS Conference Proceedings. Vol. 37.

(32) SCOTT D., STRACHEY C.

Towards a mathematical semantics for computer languages.
Symposium on Computers and Automata. 1971.
Polytechnic Institute of Brooklyn.

(33) STRACHEY C.

A general purpose macro generator.
Computer Journal. Octobre 1965.

(34) STRACHEY C.

Varieties of programming languages.
International Computing Symposium.
Venise. Avril 1972.

(35) VAN WIJNGAARDEN A., MAILLOUX B., PECK J., KOSTER C.

Report on the algorithmic language ALGOL 68.
Second Printing. Mathematisch Centrum Amsterdam.
MR 101 Octobre 1969.

(36) VIDART J.

Macros syntaxiques et langages de programmation "système".
Centre Scientifique IBM, Grenoble.
Etude FF2-0150. Janvier 1973.

(37) WAITE W.M.

LIMP. A language independant macro processor.
Communications of A.C.M. Juillet 1967.

(38) WILLIS B.

Définition et implantation de la sémantique des langages de programmation.
Thèse de Doctorat d'Etat. Mars 1974.
Université S. et M. de Grenoble.

(39) WIRTH N., HOARE C.A.R.

A contribution to the developement of ALGOL.
Communications of A.C.M. Juin 1966.

(40) WIRTH N.

Program development by stepwise refinement.
Communications of A.C.M. Avril 1971.

(41) WIRTH N.

The programming language PASCAL.
Revised Report.
International Summer School on Structured programming and Programmed Structures.
Munich 1973.