N° d'ordre :

THESES

présentées à

LA FACULTE DES SCIENCES DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR ES SCIENCES APPLIQUEES

par

J. Cohen

Première thèse :

LANGAGES POUR L'ECRITURE DE COMPILATEURS

Deuxième thèse :

PROPOSITIONS DONNEES PAR LA FACULTE

Thèses soutenues le juin 1967, devant la Commission d'examen

MM. J. KUNTZMANN, Président

N. GASTINEL
B. VAUQUOIS
L. BOLLIET
J. C. BOUSSARD

examinateurs.

F A C U L T E D E S S C I E N C E S

LISTE DES PROFESSEURS

DOYENS HONORAIRES :

M. MORET

M. WEIL

DOYEN :

M. BONNIER E.

PROFESSEURS TITULAIRES :

MM.	NEEL Louis	Chaire de Physique Expérimentale
	HEILMANN René	Chaire de Chimie
	KRAVTCHENKO Julien	Chaire de Mécanique Rationnelle
	CHABAUTY Claude	Chaire de calcul différentiel et intégral
	BENOIT Jean	Chaire de Radioélectricité
	CHENE Marcel	Chaire de Chimie Papetière
	WEIL Louis	Chaire de Thermodynamique
	FELICI Noël	Chaire d'Electrostatique
	KUNTZMANN Jean	Chaire de Mathématiques Appliquées
	BARBIER Reynold	Chaire de Géologie Appliquée
	SANTON Lucien	Chaire de Mécanique des Fluides
	OZENDA Paul	Chaire de Botanique
	FALLOT Maurice	Chaire de Physique Industrielle
	KOSZUL Jean-Louis	Chaire de Mathématiques M.P.C.
	GALVANI O.	Mathématiques
	MOUSSA André	Chaire de Chimie Nucléaire
	TRAYNARD Philippe	Chaire de Chimie Générale

	SOUTIF Michel	Chaire de Physique Générale
	CRAYA Antoine	Chaire d'Hydrodynamique
	REULOS R.	Théorie des Champs
	BESSON Jean	Chaire de Chimie
	AYANT Yves	Physique Approfondie
	GALLISSOT	Mathématiques
Melle	LUTZ Elisabeth	Mathématiques
MM.	BLAMBERT Maurice	Chaire de Mathématiques
	BOUCHEZ Robert	Physique Nucléaire
	LLIBOUTRY Louis	Géophysique
	MICHEL Robert	Chaire de Minéralogie et Pétrographie
	BONNIER Etienne	Chaire d'Electrochimie et d'Electrométallurgie
	DESSAUX Georges	Chaire de Physiologie Animale
	PILLET E.	Chaire de Physique Industrielle et Electrotechnique
	YOCCOZ Jean	Chaire de Physique Nucléaire Théorique
	DEBELMAS Jacques	Chaire de Géologie Générale
	GERBER R.	Mathématiques
	PAUTHENET R.	Electrotechnique
	VAUQUOIS B.	Chaire de Calcul Electronique
	BARJON R.	Physique Nucléaire
	BARBIER Jean-Claude	Chaire de Physique
	SILBER R.	Mécanique des Fluides
	BUYLE-BODIN Maurice	Chaire d'Electronique
	DREYFUS B.	Thermodynamique
	KLEIN J.	Mathématiques
	VAILLANT F.	Zoologie et Hydrobiologie
	ARNOUD Paul	Chaire de Chimie M.P.C.
	SENGEL P.	Chaire de Zoologie
	BARNOUD F.	Chaire de Biosynthèse de la Cellulose
	BRISSONNEAU P.	Physique
	GAGNAIRE Didier	Chaire de Chimie Physique

Mme	KOFLER L.	Botanique
MM.	DEGRANGE Charles	Zoologie
	PEBAY-PEROULA J.C.	Physique
	RASSAT A.	Chaire de Chimie Systématique

PROFESSEURS SANS CHAIRE :

MM.	GIDON P.	Géologie et Minéralogie
	GIRAUD P.	Géologie
	PERRET R.	Servomécanismes
Mme	BARBIER M.J.	Electrochimie
Mme	SOUTIF J.	Physique
MM.	COHEN J.	Electrotechnique
	DEPASSEL R.	Mécanique des Fluides
	GASTINEL N.	Mathématiques Appliquées
	ANGLES-d'AURIAC P.	Mécanique des Fluides
	DUCROS P.	Minéralogie et Cristallographie
	GLENAT R.	Chimie
	LACAZE A.	Thermodynamique
	BARRA J.	Mathématiques Appliquées
	COUMES A.	Electronique
	PERRIAUX J.	Géologie et Minéralogie
	ROBERT A.	Chimie Papetière
	BIAREZ J.P.	Mécanique Physique
	BONNET G.	Electronique
	CAUQUIS G.	Chimie Générale
	BONNETAIN L.	Chimie Minérale
	DEPOMMIER P.	Etude Nucléaire et Génie Atomique
	HACQUES Gérard	Calcul Numérique
	POLOUJADOFF M.	Electrotechnique

MAITRES DE CONFERENCES :

MM.	DODU J.	Mécanique des Fluides
	LANCIA Roland	Physique Automatique
Mme	KAHANE J.	Physique
MM.	DEPORTES C.	Chimie
Mme	BOUCHE L.	Mathématiques
MM.	SARROT-RAYNAUD J.	Géologie Propédeutique
Mme	BONNIER M.J.	Chimie
MM.	KAHANE A.	Physique Générale
	DOLIQUE J.M.	Electronique
	BRIERE G.	Physique M.P.C.
	DESPRE P.	Chimie S.P.C.N.
	LAJZEROWICZ J.	Physique M.P.C.
	VALENTIN P.	Physique M.P.C.
	BERTRANDIAS J.P.	Mathématiques Appliquées T.M.P.
	LAURENT P.	Mathématiques Appliquées T.M.P.
	CAUBET J.P.	Mathématiques Pures
	PAYAN J.J.	Mathématiques
Mme	BERTRANDIAS F.	Mathématiques Pures M.P.C.
MM.	LONGEQUEUE J.P.	Physique
	NIVAT M.	Mathématiques Appliquées
	SOHM J.C.	Electrochimie
	ZADWORNY F.	Electronique
	DURAND F.	Chimie Physique
	CARLIER G.	Biologie Végétale
	AUBERT G.	Physique M.P.C.
	DELPUECH J.J.	Chimie Organique
	PFISTER J.C.	Physique C.P.E.M.
	CHIBON P.	Biologie Animale
	IDELMAN S.	Physiologie Animale
	BLOCH D.	Electrotechnique
	BRUGEL L.	I.U.T.
	SIBILLE R.	I.U.T.

Monsieur le Professeur J. KUNTZMANN a bien voulu s'intéresser à mon travail et me fait l'honneur de présider la commission d'examen. Je le prie d'accepter mes remerciements très sincères.

Je remercie également Messieurs les Professeurs N. GASTINEL et B. VAUQUOIS qui ont bien voulu participer à ce Jury.

AVANT-PROPOS

Ce travail représente un résumé des études que j'ai eu le plaisir de faire pendant mon séjour de quatre ans à Grenoble. Sans le dynamisme et la gentillesse de mes amis de l'Institut de Mathématiques Appliquées ce travail n'aurait jamais vu le jour.

A la fin de 1963 Louis BOLLIET m'aiguilla vers les problèmes de construction des compilateurs. Son intérêt pour ce sujet qui m'était inconnu fut contagieux. C'est grâce à lui que mon séjour à Grenoble n'a pas été seulement intéressant sur le plan professionnel mais encore du point de vue humain. Les voyages et les stages que j'ai eu la chance de faire sur sa suggestion et grâce à son appui m'ont donné une occasion unique d'étudier les problèmes de la compilation.

Mon travail à Grenoble a pratiquement commencé par une étude critique des méthodes d'analyse syntaxique. J'ai bénéficié de la collaboration précieuse de Monique BRASSEUR, qui m'a initié aux études plus théoriques des langages. C'est avec son aide que j'ai composé une grande partie du paragraphe IV.1. Peu après j'ai étudié avec la collaboration de NGUYEN Dinh Xuan l'effet de l'ordonnancement des règles dans l'analyse syntaxique (cf. § IV.3).

A la suite de cette période d'initiation aux problèmes de compilation et sur la suggestion de Louis BOLLIET j'ai eu l'occasion de me familiariser avec le langage Lisp et de réaliser avec l'aide de

NGUYEN Huu Dung une inclusion de Lisp dans Algol (cf. § V.1). Habitué comme je l'étais aux langages "à une dimension" tels qu'Algol, la découverte d'une nouvelle dimension apportée par la récursivité de Lisp m'a laissé à la fois étonné et conquis. J'ai pu partager cet enthousiasme avec Yvon SIRET et Laurent TRILLING qui ont apporté des améliorations considérables au système Lisp-Algol.

Depuis Mars 1965 j'ai eu l'occasion de travailler en contact plus étroit avec Jean-Claude BOUSSARD. Son objectivité, son sens critique et l'expérience qu'il a acquise dans la réalisation d'un compilateur Algol m'ont été d'une importance capitale pour la mise au point du langage d'édition. C'est avec lui et Philippe JORRAND que j'ai préparé une grande partie du matériel du chapitre III.

Durant l'année dernière j'ai eu comme camarade de bureau Laurent TRILLING. Je dois avouer qu'il serait difficile d'en trouver un plus courtois, patient et enthousiasmé par les idées précipitées qui parfois me venaient à la tête. Le travail décrit au chapitre VI et dans l'Appendice B a été réalisé avec sa collaboration.

Tous les programmes Algol présentés dans le texte ont été mis au point au moyen du compilateur de Grenoble. A plusieurs reprises Alain AUROUX nous a accordé des passages de programme supplémentaires et il nous a aussi expliqué quelques-unes des particularités du 7044 en ce qui concerne l'unité de disques ; il est l'auteur des programmes en code mentionnés en Appendice A.

La programmation étant encore dans l'enfance, beaucoup "d'astuces" sont transmises au cours des bavardages entre collègues. C'est avec Jean du MASLE et Jean Le PALMEC que j'ai eu l'occasion de discuter et d'apprendre quelques-uns de ces tours de mains si importants dans la construction des compilateurs.

En ce qui concerne la rédaction de ce travail j'avoue qu'après avoir en vain fouillé le Manuel du Style, essayé la lecture de Mérimée et d'autres classiques, j'ai décidé tout simplement de demander l'avis objectif de mon professeur de français et collègue Olivier LECARME. L'élimination des redondances et complexités est sa spécialité et il a en plus réussi à enlever tous les brésilianismes et anglicismes de la version originale. Ceux qui restent sont de mon exclusive responsabilité, comme cette dernière tournure.

Monsieur le Professeur N. GASTINEL, Jean-Claude BOUSSARD, Jean Le PALMEC, Olivier LECARME, Laurent TRILLING et Georg WERNER ont lu le manuscrit original. Après avoir fait l'union de leurs remarques je pensais qu'il serait inutile de présenter cette thèse. Heureusement, c'est leur intersection qui a, je l'espère, rendu ce travail moins pénible à lire.

Michela COOPER a dactylographié l'ébauche et la version finale de ce travail avec le soin et l'efficacité américains. Messieurs MOUNET, VIDAL et DEBARGUES ont permis la réalisation matérielle de cet ouvrage.

TABLE DES MATIERES

I. Introduction..... 1

1. Généralités..... 1

2. Etude critique des travaux existants..... 8

3. Objectif..... 16

II. Justification de la stratégie adoptée..... 18

III. Langage d'édition..... 23

1. Introduction..... 24

2. Syntaxe et sémantique..... 32

3. Exemple d'utilisation..... 41

4. Réalisation pratique..... 43

5. Transformation des règles sous FNB en tables d'états..... 48

IV. Analyse..... 56

1. Analyse syntaxique..... 58

2. Description de quelques méthodes d'analyse.... 70

3. Mise en ordre des règles syntaxiques..... 106

4. Remarques sur l'analyse..... 111

5. Langage d'analyse..... 118

V. Langage pour la manipulation des arbres..... 120

1. Procédures de base (première version)..... 121

2. Deuxième version..... 147

3. Remarques sur l'utilisation des procedures.... 150

4. Représentation des arbres..... 154

5. Exemples..... 156

VI. Génération..... 172

1. Exemple d'utilisation du résultat de l'analyse..... 175

2. Langage de génération..... 179

3. Description d'un transducteur..... 194

VII. Mise en oeuvre d'un système complet..... 205

1. Système minimal..... 206

2. Système idéal..... 209

VIII. Conclusions..... 213

Appendice

A. Utilisation des mémoires secondaires..... 221

B. Remarques sur la récupération quand on utilise les mémoires de masse..... 232

Bibliographie..... 240

I. Introduction

"L'art de raisonner se réduit
à une langue bien faite."
(Condillac, Oeuvres)

"Je vis de bonne soupe et non
de beau langage."
(Molière, Les Femmes Savantes)

I.1. Généralités

La prolifération des langages de programmation, qu'elle soit un mythe ou une réalité, a, en tout cas, servi à diriger l'attention des utilisateurs des machines électroniques vers l'étude des moyens de construction rapide des compilateurs. Nous trouvons actuellement trois voies possibles pour surmonter les difficultés posées par la construction rapide des compilateurs. Ce sont :

A. Inclusion ("embedding") dans un langage hôte

Au lieu de créer de nouveaux langages, on choisit un langage "réceptif" dans lequel on essaie d'exprimer les opérations voulues. Nous entendons par langage hôte "réceptif" un langage qui possède au moins la notion de procédures (Algol) ou de sous-programmes (Fortran). Les instructions du "nouveau" langage sont faites d'appels à quelques procédures de base ou à d'autres procédures qui les utilisent. Cette méthode a été employée par WEIZENBAUM pour donner à Fortran la possibilité de

manipuler les listes [We]*. Nous-mêmes avons écrit un jeu de procédures qui permettent d'inclure Lisp dans Algol ([CD] et § V.1). Algol en effet est un hôte particulièrement réceptif à Lisp du fait que l'on peut appeler récursivement les procédures. D'autres inclusions de langages dans Lisp et Fortran sont décrites en [BW].

L'inclusion est sans doute la voie la plus rapide pour l'établissement de nouveaux langages, tout simplement parce qu'elle évite la construction de compilateurs spéciaux. Parfois l'hôte n'est pas très réceptif ; une greffe plus profonde est nécessaire : elle consiste à modifier le compilateur de façon à ce qu'il accepte de nouvelles instructions. Des changements profonds ont été par exemple effectués dans un compilateur Algol pour pouvoir traiter efficacement les problèmes non strictement séquentiels requis par les langages de simulation [DN]. Il reste que même avec des changements plus profonds (adaptés en général à un langage précis), l'inclusion est loin d'être une solution idéale puisqu'elle reste limitée par la syntaxe du langage hôte. Peut-être l'adoption de langages offrant les ressources de "macro-instructions" rendra-t-elle l'inclusion plus intéressante qu'elle ne l'est aujourd'hui.

B. Ecriture d'un compilateur dans un langage évolué

Remarquons d'abord que cette voie est la plus naturelle si l'on considère l'évolution progressive des assembleurs vers les compilateurs. Ces derniers sont à l'heure actuelle presque toujours écrits en langage

---

(*) L'abbréviation contenue entre deux crochets se rapporte à la bibliographie.

d'assemblage : pourquoi ne pas les écrire en utilisant un langage plus évolué ? La réponse à cette question est liée à des facteurs à la fois techniques et économiques. La tendance de la technologie est d'alléger le travail de l'homme même au prix d'un gaspillage de la machine, mais cela ne se fait que lorsque le coût de la machine devient inférieur à celui de l'effort humain qu'elle économise. On a justement fait les compilateurs pour réduire l'effort de l'homme tout en sachant qu'ils généreraient un code moins efficace que celui produit par un programmeur compétent (surtout dans le cas de langages tels qu'Algol). Le succès des compilateurs est plus évident dans le cas où les programmes générés ne sont exécutés qu'un nombre limité de fois ou lorsqu'ils sont soumis à des changements fréquents. Or les compilateurs sont des programmes exécutés très souvent et auxquels on apporte rarement des modifications, une fois qu'ils ont été mis au point. Ces caractéristiques des compilateurs les situent donc à l'extrême opposé des programmes qu'ils génèrent. Afin d'obtenir plus rapidement des compilateurs on peut évidemment sacrifier la place et la vitesse en les écrivant dans un langage plus évolué. Cependant, à l'heure actuelle, l'addition de quelques milliers de mémoires et l'augmentation de vitesse d'une machine coûtent probablement plus cher que la construction d'un compilateur en langage d'assemblage par une équipe de programmeurs. Une telle situation peut évidemment se renverser si le coût des machines devient inférieur à celui des programmes. Pour justifier notre point de vue, mentionnons les efforts réalisés par l'équipe de Grenoble pour décrire en Algol un compilateur Algol. Dans le

travail présenté en [BC 1] décrivant un compilateur simplifié qui ne traitait que des variables entières, nous sommes arrivés à la limite des programmes compilables sur le 7044. Jean du MASLE [dM 1] a poussé beaucoup plus loin cet effort par la réalisation complète d'un compilateur Algol en Algol selon la méthode décrite en [RR]. Ce compilateur en deux passages génère des programmes en langage intermédiaire ; il est divisé en plusieurs maillons et utilise toutes les ressources de l'unité de disques. Cependant son emploi effectif est impraticable à cause de sa lenteur.

Malgré ce que l'on vient de dire il existe trois cas qui justifient l'écriture d'un compilateur dans un langage plus évolué. Le premier cas est celui des langages dont la syntaxe est simple. Par exemple, le programme interpréteur de Lisp peut être décrit de façon admirablement concise en Lisp ([Mc 1, Mc 2] et § V.5.2). D'autres langages tels que Neliac [Hal] ont été développés au moyen de compilateurs de plus en plus puissants écrits en Neliac ("bootstrapping"*). Le deuxième cas est celui de l'écriture d'un compilateur en langage plus évolué pour définir rigoureusement la sémantique du langage : la description d'un compilateur Algol en Algol définit de façon précise le fonctionnement du compilateur, beaucoup plus rigoureusement qu'un rapport sur le langage. Un troisième avantage apporté par l'écriture d'un compilateur en langage plus évolué

(*) Notre expérience avec Algol nous rend sceptiques sur l'efficacité de cette technique et sur l'encombrement qui en résulte.

est qu'il permet de mettre au point les algorithmes complexes de compilation et rend donc plus facile le travail de traduction en langage d'assemblage. (C'est d'ailleurs avec cette optique que nous présentons ici les algorithmes de compilation en Algol.) Finalement, l'aspect didactique d'une telle description n'est pas lui non plus négligeable.

Passons maintenant à la critique la plus sévère que l'on puisse faire à ce moyen de construction rapide des compilateurs : il ignore complètement les parties communes qui peuvent exister lorsqu'on écrit plusieurs compilateurs pour plusieurs langages. Si en effet il existe des facteurs communs à plusieurs langages - et c'est là la question cruciale ! - cette voie ne réussit pas à les mettre en évidence.

C. Paramétrisation des compilateurs

Les recherches visant à une mise en facteur des propriétés communes aux langages de programmation ont partagé les auteurs de compilateurs en deux groupes :

1. Le premier a adopté le principe de la description des langages sous forme normale de Backus [Bac]. Cette forme de description avait été employée initialement pour définir la syntaxe d'Algol [Na 1, Na 2], et par la suite pour décrire plusieurs autres langages. L'intérêt de cette voie a augmenté lorsqu'on a établi un lien entre la forme normale de Backus (FNB) et la théorie des langages proposée par Chomsky [Ch 1]. Ceci doit pouvoir aider à réduire l'empirisme qui prédomine dans la compilation, en la

fondant sur une base mathématique solide. L'argument principal contre cette idée est l'impossibilité de décrire complètement les langages de programmation au moyen de la forme normale [Fl 1]. Cependant, l'expérience scientifique passée nous montre que la plupart des modèles mathématiques (en physique, par exemple) doivent être appliquées judicieusement sous peine de mener à des résultats contradictoires. Le plus souvent ces derniers s'expliquent au moyen d'un nouveau modèle qui englobe le précédent. Ainsi la formule classique Force = masse x accélération, valable et utile dans un très grand nombre de cas, ne l'est pas pour les très grandes accélérations. Vue comme un modèle, la description des langage sous FNB permet de mettre en évidence la syntaxe d'un grand nombre de langages de programmation. Mais la syntaxe n'est malheureusement qu'une partie de la définition d'un langage et la mise en facteur reste toujours incomplète. Trois pionniers se détachent parmi les chercheurs qui ont exploré cette voie : Irons écrivit en 1961 son compilateur dirigé vers la syntaxe [Ir 1, Ir 3], à peu près vers l'époque où Brooker et Morris travaillaient sur leur compilateur de compilateurs [BM, BMMR].

2. Le deuxième groupe a pris une attitude plutôt pragmatique vis-à-vis de la syntaxe : elle ne sert, pensent-ils, qu'à définir partiellement un langage, mais la compilation est trop complexe

pour qu'on puisse utiliser cette syntaxe comme élément important de mise en facteur des langages. Cette mise en facteur, affirment-ils, ne peut être réalisée que de façon empirique : on fournit à l'utilisateur les moyens de reconnaissance des structures imbriquées, des ordres spéciaux pour les recherches en table, pour les manipulations de bits, la génération des instructions de la machine, etc. [01]. FELDMAN [Fe 1, Fe 2] a proposé un langage d'écriture des compilateurs où la reconnaissance se fait par un algorithme de Markov suggéré par FLOYD [Fl 2]. GARWICK [Gar 1] en revanche utilise la récursivité pour reconnaître les structures parenthésées des langages du type Algol.

Dans ce travail nous nous plaçons surtout dans le premier groupe. Cependant, une différence importante nous en écarte : nous proposons une phase de préparation initiale du programme-source afin de le rendre plus adapté à une analyse à faire au moyen de la description sous FNB du langage que l'on désire traiter.

I.2. Etude critique des travaux existants

Dans ce paragraphe nous nous bornerons à un bref tour d'horizon sur les langages d'écriture des compilateurs. Nous reportons au chapitre IV la description détaillée de quelques-unes des méthodes d'analyse qu'ils utilisent.

IRONS a proposé en 1961 un compilateur dirigé par la syntaxe, décrit en [Ir 1, Ir 3, Ir 4]. Plusieurs des idées qu'il a introduites dans son compilateur sont parmi les plus intéressantes dans ce domaine. Irons a écrit son algorithme de reconnaissance sous forme récursive ; nous verrons au chapitre IV que cette forme d'écriture se prête bien à l'analyse des langages dont la syntaxe se décrit sous FNB. De plus, il a choisi un algorithme d'analyse sélective, c'est-à-dire capable d'éviter certaines impasses rencontrées dans le processus d'essais et d'erreurs utilisé pendant la reconnaissance : quelques années plus tard GRIFFITHS et PETRICK [GP] ont montré que cet algorithme était parmi les plus efficaces. La façon de traiter les déclarations dans les langages du type d'Algol a été elle aussi une des idées originales d'IRONS : quand l'analyseur reconnaît une déclaration il cède momentanément le contrôle au générateur, lequel ajoute des règles à la grammaire, règles qui sont utilisées ensuite dans l'analyse des instructions du programme. Cette génération de nouvelles règles rend l'analyseur d'IRONS capable de traiter des langages plus complexes que ceux qui s'expriment sous FNB. Quant à la détection des erreurs, IRONS a proposé l'introduction de règles

factices qui permettent la reconnaissance des erreurs les plus fréquentes*.

Au sujet de la génération, le formalisme proposé par IRONS permet de produire un programme en langage d'assemblage à partir de la configuration de l'arbre d'analyse syntaxique. (La méthode que nous proposons au chapitre IV est une simplification de ce formalisme.) Le plus grand reproche que l'on puisse faire aux travaux d'IRONS est qu'il a voulu tout exprimer d'une façon condensée en fonction de la FNB. Son compilateur ne dispose pas des moyens élémentaires nécessaires à une analyse lexicographique préalable d'un programme : il opère en un seul passage utilisant un analyseur assez puissant même pour reconnaître les identificateurs et les nombres. Remarquons pour terminer que, la génération étant étroitement liée à l'analyse, le formalisme d'IRONS ne permet pas la vue d'ensemble nécessaire à la génération de programmes objets efficaces. Nous ne croyons pas que le compilateur d'IRONS ait jamais été opérationnel. Cependant le groupe de l'Université de Cambridge en Angleterre a utilisé cette méthode pour écrire le compilateur CPL [Bar] sur Atlas. Ce dernier est opérationnel.

(*) Observons que ceci peut rendre le langage ambigu. Ainsi, en Algol, peut-on, si l'on oublie un délimiteur début ou fin, avoir plusieurs façons de placer l'élément qui manque. Les algorithmes d'analyse multiple pour les langages contenant des ambiguités seront mentionnés au chapitre IV (par exemple, cf. [Ir 2]).

BROOKER et MORRIS [BM, Mor] ont réalisé leur compilateur de compilateurs vers la même époque que celui d'IRONS. Notre première critique concerne les publications qui décrivent leur compilateur : elles manquent de la clarté et de la continuité nécessaires à la compréhension de l'ensemble des travaux ; les meilleures descriptions de ce compilateur ont été faites par d'autres chercheurs [Ro, dM 2] qui se sont intéressés à leur projet. Le compilateur de BROOKER et MORRIS accepte la syntaxe d'un langage source sous une FNB modifiée mais qui n'en est pourtant pas plus puissante. Les modifications consistent en l'introduction des symboles * et ? pour indiquer respectivement la concaténation et l'emploi facultatif des métavariables(*). Les règles de syntaxe du langage que l'on désire traiter se divisent en deux groupes : le premier - celui des "phrases" - définit les métavariables auxquelles on n'associe pas de génération, et le deuxième - celui des "formats" - définit les métavariables auxquelles sont associés des sous-programmes de génération. Ces sous-programmes, écrits en langage d'assemblage, sont eux-mêmes décrits sous FNB par des phrases et des formats ; les sous-programmes associés à ces derniers formats sont en langage machine. L'analyseur du langage source est donc utilisé aussi dans la reconnaissance des sous-programmes associés aux formats. La correspondance entre les éléments analysés et les éléments qu'on doit générer se fait comme pour les paramètres effectifs et formels

---

(*) Le symbole * a la même signification que celui utilisé dans les expressions régulières.

en Algol : les sous-programmes comportent des paramètres formels qui seront remplacés par les paramètres effectifs fournis par l'analyseur. Nous verrons au chapitre IV que la méthode d'analyse proposée par BROOKER et MORRIS est moins efficace que celle d'IRONS ; de plus, cette méthode ne permet pas d'avoir dans la grammaire de règles récursives à gauche. En revanche, BROOKER et MORRIS accordent à l'utilisateur une plus grande liberté d'insertion de ses points de génération, ce qui permet de produire des programmes objets plus efficaces. Mais il reste qu'une grande partie des sous-programmes en code ont été incorporés au fur et à mesure des besoins de compilation pour différents langages ; l'utilisation du système de BROOKER et MORRIS est devenue donc très dépendante de la machine Atlas sur laquelle il a été implanté. Le compilateur de BROOKER et MORRIS est opérationnel ; il a été utilisé pour construire plusieurs compilateurs pour Fortran et d'autres langages du type Algol.

Le langage proposé par FELDMAN [Fe 1, Fe 2] utilise la syntaxe uniquement comme guide d'écriture des compilateurs. Les instructions de son langage sont définies sous FNB de la façon suivante :

⟨instruction⟩ ::= ⟨étiquette⟩ :
⟨sommet de pile⟩ → ⟨nouveau sommet⟩ ||
⟨action⟩ || ⟨prochaine étiquette⟩

où : → et || sont des séparateurs ; les métavariables ⟨sommet de pile⟩ et ⟨nouveau sommet⟩ sont des séquences de symboles, et une ⟨action⟩ peut se définir comme un appel à une procédure, pour générer du code machine,

pour faire des recherches en table, pour lire un symbole, etc. ; <étiquette> et <prochaine étiquette> ont la même signification qu'une <étiquette> en Algol. Un programme écrit dans le langage de FELDMAN est formé d'une séquence de telles instructions. Le compilateur du langage défini par l'utilisateur emploie une pile explicite ; l'algorithme de compilation consiste à comparer successivement le sommet de cette pile avec chaque <sommet de pile> des instructions spécifiées par l'utilisateur. Lorsqu'une comparaison est satisfaite le sommet de la pile est remplacé par la séquence indiquée par <nouveau sommet> ; l'<action> est exécutée et le contrôle passe à la <prochaine étiquette>. Cet algorithme dit du type de Markov a été proposé par FLOYD en [F1 2]. L'analyseur de FELDMAN est déterministe : il exécute toujours la première instruction applicable dans une suite d'instructions, même s'il en existe dans cette suite d'autres également applicables. FELDMAN affirme sans le démontrer qu'il existe une équivalence entre cet algorithme et les méthodes d'analyse classiques comme celles d'IRONS ou BROOKER et MORRIS. Il mentionne - sans le décrire - un algorithme capable de générer un programme dans son langage à partir de la description sous FNB d'un langage donné. Il n'utilise pas cet algorithme parce que les actions introduites n'ont parfois rien à voir avec l'analyse. Signalons à ce propos qu'il se manifeste actuellement parmi les chercheurs qui s'intéressent à l'analyse syntaxique une tendance à développer des algorithmes de ce type ; leurs recherches se dirigent vers les sous-ensembles des langages que l'on peut décrire sous FNB et pour lesquels les instructions du type utilisé par FELDMAN peuvent

être automatiquement générées par algorithme. C'est ainsi que FLOYD a défini les langages "bounded context" [Fl 3] et que plus récemment KNUTH [Kn] a généralisé ce concept avec les langages LR(k). Le compilateur de FELDMAN fonctionne en un seul passage ; il est opérationnel sur le G-20 du Carnegie Institute of Technology. Il a été utilisé pour l'enseignement des techniques de compilation, pour la construction d'un compilateur Algol, et pour l'implantation du langage Formula-Algol de manipulation des expressions algébriques [PIS].

GARWICK [Gar 1] a proposé un langage qui, comme celui de FELDMAN, n'utilise pas directement les règles de syntaxe sous FNB. Dans ce langage un programme est formé de plusieurs groupes d'instructions, chacune pouvant être décrite par la "suite" Algol :

```
<étiquette> : si <expression booléenne> alors
                                          début
                                             <action> ;
                                             allera <étiquette>
                                          fin
                                        sinon allera <étiquette>
```

Un groupe de ces instructions se comporte comme une procédure au sens d'Algol. L'<expression booléenne> vérifie si le dernier symbole lu est d'un certain type ou si certains registres utilisés par le programme généré sont déjà occupés. L'<action> est exécutée lorsque l'expression booléenne est vraie : elle consiste à lire des symboles, générer une instruction machine ou appeler d'autres groupes d'instructions, les

appels récursifs étant évidemment nécessaires. En opposition avec la technique de FELDMAN la pile de récursivité chez GARWICK est implicite et inaccessible à l'utilisateur. Une analogie pourra rendre clair le fonctionnement du langage de GARWICK : il s'apparente à celui d'un compilateur écrit en Lisp pour les langages du genre Algol. Une caractéristique intéressante du compilateur de GARWICK est son système d'allocations de tables pendant la compilation ; ce système est décrit en [Gar 2] et a été utilisé par nous dans la réalisation du langage défini au chapitre III. Un compilateur du langage de GARWICK vient d'être réalisé sur une CDC 3600 ; il doit être opérationnel actuellement mais nous ne connaissons pas encore de langages pour lesquels il ait été utilisé.

Le critique que nous faisons aux langages de FELDMAN et de GARWICK est qu'ils abandonnent l'emploi direct de la FNB, simple et bien connue, en la remplaçant par d'autres formalismes plus complexes et plus particuliers. Nous ne doutons pas que les compilateurs correspondants soient plus efficaces que ceux d'IRONS ou de BROOKER et MORRIS ; mais il est tout de même regrettable que les débutants dans ces langages soient forcés de devenir très tôt des spécialistes, la lecture des compilateurs leur étant réservée.

D'autres travaux sur les compilateurs dirigés par la syntaxe sont décrits en [BF, Bas, In 1, In 2, Wa 1, CS, LW, Le, CL, Me].

Cette liste est trop importante pour que nous puissions les aborder ici, même d'une façon sommaire. Signalons seulement que parmi ces compilateurs se détachent ceux mis en oeuvre par le groupe Computer Associates ; celui-ci a été parmi les premiers en Amérique à proposer et étudier des compilateurs de ce genre. Un analyseur du type suggéré par BROOKER et MORRIS y est utilisé, la génération se faisant à partir de la configuration de l'arbre d'analyse syntaxique.

L'abondance des publications, le nombre croissant de compilateurs opérationnels, l'étendue de la gamme de langages pour lesquels ils sont employés, témoignent de l'importance et de l'intérêt de ce sujet.

I.3. Objectif

Le but de ce travail est de présenter trois langages facilitant, compte tenu de l'état actuel des machines et du développement de la théorie des langages, l'écriture rationnelle des compilateurs. Ces langages ont été conçus pour aider à la mise au point rapide de langages expérimentaux. De même que la réalisation d'un compilateur encourage le programmeur à retoucher et perfectionner ses programmes, de même nous espérons que l'existence de langages d'écriture de compilateurs encouragera ceux qui créent des langages à les essayer et les améliorer (comme font, par exemple, les hydrauliciens avec leurs modèles).

Ayant dû nous-mêmes réaliser un effort d'expérimentation analogue, nous avons choisi Algol comme langage d'essai, de mise au point et de communication des algorithmes, surtout parce que nous pouvions accéder facilement à un bon compilateur Algol : la réalisation d'un système pratique et efficace a été remise à une étape ultérieure qui, nous l'espérons, bénéficiera des expériences et des programmes exposés ici. Cependant, il nous a paru intéressant d'ajouter au texte quand c'était possible quelques commentaires sur la réalisation efficace des programmes en langage-machine.

En même temps que cet objectif général, nous nous sommes efforcés d'en poursuivre un autre non moins important : généraliser la notion de paramétrisation, de façon à ce qu'elle concerne non seulement les

propriétés communes aux langages les plus évolués, mais aussi celles des langages-machine. Nous proposons à cet effet une nouvelle voie qui, quoique encore primitive, peut conduire à des résultats intéressants pour la description des langages-machine et des langages d'assemblage.

La mise en oeuvre de ces trois langages nous a amené à étudier plusieurs problèmes dont certains semblaient s'écarter du sujet originel. On trouvera la description de ces derniers en Appendice ; ils se rapportent à l'utilisation des mémoires secondaires et à son effet dans la récupération des données inutiles (c'est-à-dire déjà utilisées et dont on n'a plus besoin). Nous les présentons ici à cause de leur importance dans le problème général de l'allocation de mémoire en compilation.

Cette étude nous a aussi permis de dégager plusieurs voies inexplorées, que nous présentons à la fin de certains paragraphes sous le titre : sujets de recherche.

Enfin, il est juste de signaler que ce travail est encore incomplet : il le sera moins dans le cadre du système conversationnel suggéré au chapitre VII.

II. Justification de la stratégie adoptée

"Pour tuer une puce il voulait obliger
Ces dieux à lui prêter leur foudre et
leur massue"

(La Fontaine, L'homme et la puce)

L'un des trois principes de base qui nous ont guidé dans la conception des langages présentés ici est de ne pas rendre un langage trop compliqué et trop puissant pour le travail qui lui est demandé, mais sans lui enlever de généralité.

Considérons d'abord de façon intuitive le fonctionnement d'un compilateur : dans une première phase, il "examine" successivement des sous-ensembles de l'ensemble des symboles d'une chaîne donnée, c'est-à-dire qu'il vérifie la nature et la correction de certaines juxtapositions de symboles et emmagasine certaines informations ; dans une deuxième phase, il produit une chaîne résultante dont certains éléments correspondent à l'information retenue lors de la première phase. Par la suite nous appellerons la première phase reconnaissance et la deuxième action ou génération.

Une caractéristique des compilateurs, importante mais difficilement mesurable, est la fréquence d'oscillation entre les phases de

reconnaissance et de génération pour un contexte donné. Nous entendons par contexte l'étendue de la chaîne donnée qu'il faut reconnaître avant de déclencher une génération. Pour illustrer cette oscillation entre reconnaissance et action considérons deux cas limites : dans le premier cas, on examine successivement chaque symbole de la chaîne donnée et c'est dans un contexte limité que l'on passe à la phase de génération. C'est le cas par exemple de la majorité des transducteurs utilisés par les théoriciens [Gi 1, Gi 2] ; dans le deuxième cas on examine toute la chaîne donnée (le contexte est donc beaucoup plus large) avant de générer la chaîne résultante.

Entre ces deux limites se trouve toute la variété de fonctionnement des compilateurs. Pour les langages complexes, les compilateurs proches du premier cas produisent rapidement un code peu efficace, puisqu'ils n'examinent la chaîne donnée que superficiellement ; en revanche ceux qui sont proches du deuxième cas peuvent générer un code efficace au prix d'une phase de reconnaissance plus minutieuse.

Remarquons que ce qui précède ne s'applique qu'aux compilateurs des langages complexes, pour lesquels l'efficacité du code généré dépend d'un examen minutieux de larges régions du programme ; de plus la capacité de la mémoire et le temps d'exécution jouent un rôle prépondérant dans le choix d'un fonctionnement optimal. Ces deux cas limites sont utiles. Dans la phase de mise au point d'un programme il est en effet souhaitable d'avoir un compilateur rapide même s'il génère un

code peu efficace ; pour l'exploitation on a intérêt au contraire à disposer d'un compilateur qui génère un code efficace, même s'il est plus lent. Notre deuxième principe de base est donc de laisser l'utilisateur espacer à son gré les oscillations entre reconnaissance et génération, de façon à permettre la construction de compilateurs recouvrant tous les modes de fonctionnement annoncés ci-dessus.

La plupart des compilateurs écrits à la main comportent plusieurs passages ; chacun d'eux comprend plusieurs oscillations entre reconnaissance et action, la chaîne produite par un passage étant la chaîne donnée au passage suivant. Cela permet de partager le travail de reconnaissance en plusieurs niveaux. Si la capacité et la vitesse d'une machine ne permettent pas les analyses minutieuses sur un large contexte, c'est cette répartition du travail qui rend possible la génération d'un programme objet efficace. Le troisième principe de base que nous avons adopté est donc de permettre à l'utilisateur de nos langages d'effectuer son travail en plusieurs passages.

L'expérience acquise avec les compilateurs écrits "à la main" nous montre que les séries de reconnaissances et d'actions se font au moins en deux niveaux, le premier étant microscopique : il a pour but de reconnaître des juxtapositions de symboles dans un contexte très limité ; il produit une première chaîne résultante formée par des profils qualitatifs qui résument la nature des éléments de la chaîne donnée. Cette phase est appelée <u>édition</u> ou analyse lexicographique.

Elle est importante pour deux raisons : tout d'abord elle utilise les moyens de reconnaissance les plus simples, c'est-à-dire qu'elle suit le premier principe mentionné dans ce chapitre ; ensuite elle prépare une chaîne donnée pour une deuxième phase de reconnaissance et d'action plus complexe.

Le langage d'édition présenté au chapitre III permet la construction rapide d'éditeurs qui utilisent un ou plusieurs passages. Il a été conçu pour permettre la génération d'une séquence de profils qualitatifs employés au deuxième niveau de reconnaissance-action. Ce dernier niveau est fondé sur la définition sous forme normale de Backus du langage que l'on désire traiter ; les symboles de base de ce langage peuvent donc être des profils qualitatifs générés lors du passage d'édition. Le traitement de ce deuxième niveau de reconnaissance-action se fait au moyen des deux langages décrits aux chapitres IV et VI. Le premier, assez proche de la forme normale de Backus, comporte de plus le mécanisme qui permet la transformation d'un arbre d'analyse syntaxique en un autre plus approprié à la génération du langage objet.

La récursivité commune à la plupart des langages décrits sous forme normale de Backus, et les transformations d'arbres indispensables dans ce deuxième niveau, nous ont conduit à un langage général de manipulation des arbres (ou listes). Conscient de l'importance de Lisp dans ce domaine, nous avons d'abord réalisé son inclusion dans Algol (chapitre V). Le langage de génération présenté au chapitre VI est

fondé sur l'existence d'un langage de traitement de listes de ce type. Le besoin d'un langage général pour la manipulation de symboles est encore plus évident lorsque l'on considère l'intérêt de la production d'un programme objet sous plusieurs formes différentes : langage d'assemblage, macros, langage intermédiaire, ou même langage directement interprétable.

En résumé, nous proposons ici trois langages qui permettent deux niveaux de réalisation du couplage reconaissance-action. Le langage d'édition contient un mécanisme simple de reconnaissance auquel on associe des actions ayant pour but la génération d'une séquence de profils qualificatifs. Le langage d'analyse et le langage de génération forment le deuxième ensemble, qui permet la reconnaissance des langages décrits sous forme normale de Backus et la génération d'un programme objet sous plusieurs formes. Nous avons volontairement fait se recouvrir partiellement ces deux niveaux de reconnaissance-action afin que l'utilisateur ait plusieurs moyens à sa disposition pour réaliser une même opération. Cette redondance sémantique augmente la puissance d'expression d'un langage.

III. Langage d'édition

"Pour bien savoir les choses,
il faut en savoir le détail."
(La Rochefoucauld, Réflexions)

"Les maisons empêchent de voir
la ville."
(Chamfort)

Ce chapitre décrit un langage qui permet de remplacer des ensembles de symboles d'un langage de programmation quelconque par une séquence de profils qualitatifs. Son mécanisme de base est la simulation d'un automate d'états finis. A chaque état on peut associer une action ayant pour but de générer le profil voulu. La caractéristique primordiale du langage décrit est sa souplesse, qui rend très facile toute modification des résultats d'une édition donnée.

III.1. Introduction

Nous appelons édition la première phase de la compilation : elle a pour but de produire une chaîne codée dont les éléments (mot-machine) contiennent un profil qualitatif qui indique la nature des symboles ou séquences de symboles rencontrés dans le programme source. Autrement dit, l'édition consiste à reconnaître et à coder les symboles d'un programme en vue de faciliter une analyse globale ultérieure.

La plupart des compilateurs existants possèdent une telle phase préliminaire [Bol, Bou, Do, LP, Na 3, RR] : elle permet en outre de :

- reconnaître les erreurs les plus grossières ;
- améliorer l'efficacité des phases ultérieures ;
- réaliser parfois un premier traitement simultanément à la lecture.

Les caractéristiques souhaitables dans un langage d'édition sont les suivantes :

- priorité de la simplicité sur la puissance, les opérations à effectuer étant élémentaires ;
- généralité et souplesse dans les changements à apporter au langage traité, aussi bien qu'aux résultats de l'édition ;
- compilation aisée générant du code efficace ;
- redondance sémantique, l'utilisateur devant disposer de plusieurs façons différentes d'expression d'un même concept.

Ces caractéristiques étant en conflit, nous nous sommes efforcés de définir un compromis où la simplicité prédomine, mais où il reste un maximum de généralité.

Examinons plus en détail les composants souhaitables dans un tel langage.

a. Déclaration de classes de caractères alphanumériques en vue d'autoriser les opérations dont ils feront l'objet.

b. Recours à un mécanisme simple et général d'analyse d'une séquence de caractères ou classes de caractères. L'intérêt, partagé avec les théoriciens, que nous avions pour un tel mécanisme nous a conduit à l'utilisation des automates d'états finis, qui sont les plus simples des processus de reconnaissance de langages. Ce choix a l'avantage de mettre à la disposition de l'utilisateur une théorie déjà explorée.

c. Tout en voulant garder ce dernier avantage nous ne devons pas oublier qu'un calculateur est infiniment plus général qu'un automate d'états finis. La construction de tables et leur exploitation constituent des manipulations dont le lecteur averti soupçonne la nécessité, et qui devront être incorporées au mécanisme précédent.

d. Le résultat de l'édition étant une chaîne codée, il faut pouvoir en générer un élément après la reconnaissance d'un sous-ensemble de symboles choisi par l'utilisateur. Cette

génération est effectuée à l'aide d'actions couplées aux états de l'automate. Ces actions sont exprimées en termes d'appels à des sous-programmes qui permettent l'exécution d'opérations élémentaires telles que mise et recherche en tables, manipulation de bits, contrôle de la nature des symboles lus, etc.

e. L'un des buts de l'édition étant la détection d'erreurs, il nous a fallu mettre à la disposition de l'utilisateur un moyen souple de construction et d'édition des diagnostics.

III.1.1. Fonctionnement de l'automate

Nous nous contenterons d'un exemple pour illustrer le fonctionnement de notre automate. Le lecteur intéressé par les aspects théoriques pourra se référer aux ouvrages cités en [Gi 1, Har, Moo].

Supposons que l'on veuille reconnaître les chaînes de la forme :

$$a\ b^m\ c^n\ p \quad , \qquad m \text{ et } n \geqslant 0 \qquad (1)$$

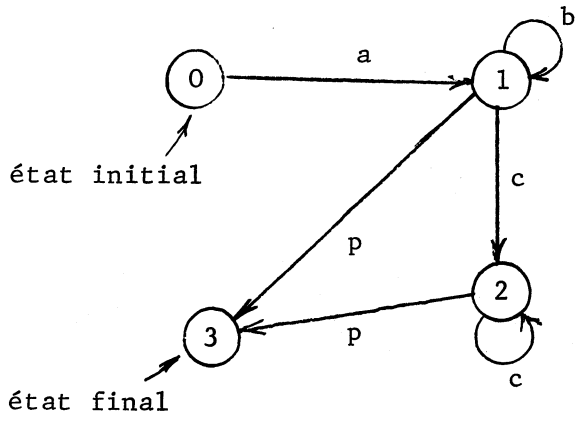
c'est-à-dire,

$$a\ \underbrace{bb\ldots b}_{m \text{ fois}}\ \underbrace{ccc\ldots cp}_{n \text{ fois}} .$$

De plus on désire produire une chaîne codée où a, b, c et p figureront par leur code (profil qualitatif), et où les entiers m et n seront insérés respectivement après b et c. Le résultat de l'édition est alors la chaîne :

code de a
code de b
entier m
code de c
entier n
code de p

Le graphe suivant représente les chemins possibles pour aboutir à la reconnaissance des chaînes (1) :



Les noeuds de ce graphe correspondent aux états 0, 1, 2 et 3 de l'automate d'états finis associé. L'état 0 est dit état initial et l'état 3 état final. Les branches ont pour nom l'un des quatre éléments a, b, c ou p. Une chaîne est dite acceptée quand en partant de l'état initial et en parcourant les branches correspondant aux symboles lus de gauche à droite on aboutit à l'état final. Ce graphe peut être représenté par la table suivante (dite table d'états de l'automate) :

	a	b	c	p
état 0	1	E	E	E
état 1	E	1	2	3
état 2	E	E	2	3
état 3				

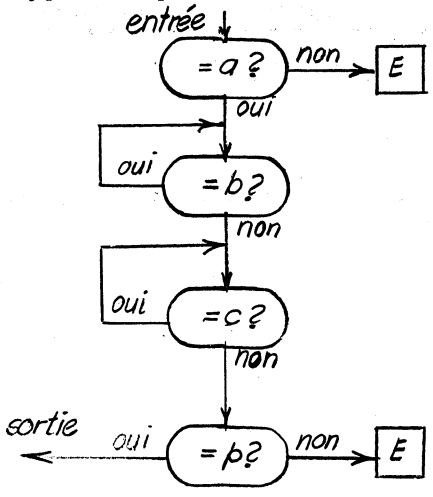
où E indique que l'apparition du symbole de la colonne correspondante est une erreur. Cela équivaut à l'introduction d'un état supplémentaire auquel toutes les branches non spécifiées dans le graphe seraient connectées.

Le procédé de reconnaissance sommairement décrit ci-dessus peut être exprimé par le programme Algol suivant :

```
retour :
état := table d'états [état, prochain symbole lu] ;          (2)
si état ≠ état final alors allera retour ;
```

Tout ceci permet donc de réaliser la reconnaissance, mais on ne dispose encore d'aucun moyen pour construire la chaîne codée.

Ouvrons maintenant une parenthèse pour établir une comparaison entre la reconnaissance par une table d'états et la programmation.
Les chaînes du type (1) peuvent être reconnues par l'algorithme suivant :



Les inconvénients d'une telle méthode résident dans son manque de souplesse et dans l'insuffisance des indications d'erreurs, à moins de complications inextricables. Ces deux inconvénients disparaissent dans la reconnaissance par l'automate d'états finis. Observons qu'il suffit de programmer un noyau représenté par le programme (2), qui sera alors valable pour toute table d'états.

Couplage des états avec des actions

Revenons maintenant au problème de la génération de la chaîne codée. On la réalise en associant à chaque état une action qui consiste à exécuter une série d'opérations spécifiées par l'utilisateur. Le programme suivant peut réaliser le couplage entre reconnaissance et génération.

```
aiguillage ACTION := A1, A2, A3, ... ;
état := 1 ; allera A1 ;
retour :                                                    (3)
état := table d'états [état, prochain symbole lu] ;
allera ACTION [état] ;

A1 : OPERATION 11 ; OPERATION 12 ; OPERATION 13 ; ...
     allera retour ;

A2 : OPERATION 21 ; OPERATION 22 ; OPERATION 23 ; ...
     allera retour ;
.
.
.
```

Par exemple, dans notre cas une OPERATION sera une procédure qui peut :

- augmenter un compteur de 1,
- générer le code de c, etc.

Dans le langage d'édition que nous présentons, l'accent a été mis sur la définition des opérations de base les plus courantes dans un tel traitement, ce qui rend la compilation des actions particulièrement simple.

III.1.2. Remarques théoriques

Il est facile d'établir un lien entre les automates d'états finis et les grammaires du type 3 de CHOMSKY [Ch 1, Ch 2]. Il suffit d'introduire des règles du type :

état	⟶	symbole	état
(ligne)		(colonne)	(élément de table)
(non terminal)		(terminal)	(non terminal)

pour obtenir les règles de grammaire correspondantes. Quand l'élément de la table représente un état final, la règle associée est terminale, c'est-à-dire que :

état	⟶	symbole
(non terminal)		(terminal)

Ainsi dans l'exemple (1) la grammaire correspondante est :

$$S_0 \longrightarrow a\ S_1$$

$$S_1 \longrightarrow b\ S_1 \quad , \quad S_1 \longrightarrow c\ S_2 \quad , \quad S_1 \longrightarrow p$$

$$S_2 \longrightarrow c\ S_2 \quad , \quad S_2 \longrightarrow p$$

où S_0 est l'élément distingué (axiome) parmi les non-terminaux.

Malheureusement les études théoriques [Ch 2] sur les langages indiquent qu'il est impossible de trouver un algorithme général qui

extraie la partie "états finis" d'un langage "context-free" donné. Il est donc impossible d'ordinaire de produire automatiquement la table d'états représentant la partie "états finis" d'un langage "context-free" général.

Nous proposons à la fin de ce chapitre une méthode qui permet, dans certains cas, d'aboutir à des résultats utiles, bien que le problème général soit insoluble.

Dans notre langage d'édition, c'est l'utilisateur qui se charge de produire la table d'états, à l'aide si possible de l'algorithme proposé au § III.5. Le couplage des états avec les actions peut parfois l'obliger à introduire des états supplémentaires.

Remarquons enfin que l'introduction parmi les opérations d'instructions manipulant une ou plusieurs piles permet l'analyse et la génération des langages "context-free" ou même plus complexes. Certains compilateurs [Na 3] ont même déjà été réalisés complètement sur ce principe.

III.2. Syntaxe et sémantique

III.2.1. Syntaxe

Un programme en langage d'édition est défini par :

<programme> ::= CLASSES : <suite de classes> *
ETATS : <suite de lignes> *
PREPARATION : <suite d'opérations> *
ACTIONS : <suite d'actions> *
ERREURS : <suite de messages d'erreurs> *

<suite de classes> ::= <classe> | <classe> ; <suite de classes>

<classe> ::= <identificateur> (<chaîne>)

<chaîne> ::= suite de caractères alphanumériques qui n'est pas une suite 01. | suite 01.>

suite 01. ::= b 01. | suite 01. 01.

<01. ::= 0 | 1.

suite de lignes> ::= <ligne> | <ligne> ; <suite de lignes>

<ligne> ::= <entier sans signe> | <entier sans signe> , <ligne>

<suite d'opérations> ::= <opération> | <opération> , <suite d'opérations>

<opération> ::= <étiquette> : <opération non étiquetée> |
<opération non étiquetée>

<opération non étiquetée> ::= <identificateur> (<liste de paramètres>)

<liste de paramètres> ::= paramètre> | <paramètre> , <liste de paramètres>

<paramètre> ::= <identificateur> | <entier> | <étiquette> | <chaîne>

<étiquette> ::= <identificateur> | FIN | RETOUR

<suite d'actions> ::= <action> | <action> ; <suite d'actions>

<action> ::= A <entier sans signe> : <suite d'opérations>

<suite de messages d'erreurs> ::= <message d'erreur> |
<message d'erreur> ;
<suite de messages d'erreurs>

<message d'erreur> ::= E <entier sans signe> : <chaîne> , <étiquette>

Les métavariables <entier>, <identificateur> et <entier sans signe> ont la même définition syntaxique (et sémantique) qu'en Algol. Le commentaire peut être introduit après un point virgule : il a pour syntaxe :

<commentaire> ::= c <chaîne>* ;

III.2.2. Sémantique

Une <classe> définit un ensemble <chaîne> que l'utilisateur choisit de grouper dans une même colonne de la table d'états de l'automate. La première <classe> d'une <suite de classes> correspond à la première colonne de la table d'états, la deuxième <classe> à la deuxième colonne, et ainsi de suite. Une <classe> peut aussi définir un profil qualitatif représenté par des <suites 01.>. On verra plus loin la signification de cette dernière métavariable. Indiquons pour le moment que la deuxième définition de <classe> n'est employée que lors d'un deuxième passage utilisant le langage d'édition. Dans les deux cas considérés précédemment, les ensembles définis par une <chaîne> ou par les différentes <chaînes> d'une <suite de classes> doivent être disjoints.

La table d'états est définie au moyen d'une <suite de lignes>, chaque <ligne> représentant une ligne de la table d'états. Le nombre d'<entiers sans signe> dans une <ligne> doit être égal au nombre de

---

(*) Ne contenant pas un ";".

<classes> dans la <suite de classes> définie préalablement. Tout <entier sans signe> dans une <ligne> doit être inférieur au nombre de <lignes> dans une <suite de lignes>.

La phase PREPARATION est faite d'une suite d'opérations d'initialisation. Ces <opérations>, dont quelques-unes seront définies plus loin, sont syntaxiquement analogues aux <opérations> de la phase ACTION. Une des opérations d'initialisation les plus fréquentes est l'affectation qui permet de donner un nom à un <entier> ou à une <suite 01.>. Cette dernière métavariable définit une suite de 0, 1 et . de longueur fixe pour un programme en langage d'édition. Cette longueur est celle du mot de la machine pour laquelle on fait l'édition. Les 0 et 1 occupent un bit et ont la signification habituelle de faux et vrai. Le point correspond à une position neutre, c'est-à-dire, qui peut être considérée comme vraie ou fausse*. Signalons que lors d'un premier passage d'édition une <suite 01.> ne contient que des 0 et 1.

L'étiquette A <entier sans signe> qui précède une <suite d'opérations> de la phase ACTION correspond à l'état indiqué par l' entier sans signe>. Ce dernier entier doit correspondre à un numéro de ligne (état) d'une <suite de lignes>. Les opérations fonctionnent

---

(*) Remarquons qu'une <suite 01.> occupe réellement deux mots de la machine, l'un pour les 0 et 1 présents dans la suite et l'autre pour indiquer les positions des points. Ainsi la suite b 00.1..10 est représentée par le couple 00010010 et 0010110.

comme des sous-programmes dont les paramètres ne peuvent être que des variables simples, c'est-à-dire <identificateur>, <entier>, <étiquette> ou <chaîne>. Les étiquettes spéciales FIN et RETOUR indiquent respectivement la fin d'un programme d'édition et un retour au fonctionnement de l'automate (cf. § III.1).

Avant de décrire quelques-unes des opérations les plus courantes, nous présentons ci-dessous les variables accessibles à l'utilisateur.

a. Les tables à une dimension T1, T2, T3 ... indicées par it1, it2, etc. Ces tables sont utilisées pour ranger les entiers ou les <suites 01.> ayant toujours une longueur fixe, égale à celle du mot de la machine. Remarquons qu'une table peut aussi fonctionner comme une pile.

b. Les magasins M1, M2, M3, ... indicés par im1, im2, etc. Ces magasins ont été conçus pour ranger des chaînes de caractères de longueur arbitraire. Chacune des chaînes rangées dans un magasin est en général suivie d'un code, c'est-à-dire d'une <suite 01.>, choisi par l'utilisateur. Les magasins peuvent avoir plusieurs niveaux : l'ouverture d'un niveau ou l'effacement du dernier niveau se font par des opérations du langage.

c. BUFFER. Le buffer est une table spéciale où sont rangés les caractères alphanumériques à mesure qu'ils sont lus. Il est

utilisé dans des opérations qui permettent de transférer son contenu dans un magasin, de comparer son contenu avec un magasin ou de le vider. Le pointeur "ib" du buffer est lui aussi accessible à l'utilisateur.

d. SYMBOLE LU. Cet entier contient le code du dernier symbole lu.

e. CLASSE. Cet entier indique la classe du symbole lu.

f. ETAT COURANT. Cet entier contient l'état courant de l'automate.

g. ie. Cet entier pointe sur le dernier mot édité.

h. On peut aussi utiliser les variables simples au sens d'Algol.

Nous présentons ci-dessous les opérations les plus fréquentes ; on en trouvera une liste complète en [Jor].

	Effets et commentaires
a. Opérations élémentaires	
AFFECTATION (V1, V2)	V1:=V2, V1 et V2 sont soit un ⟨entier⟩ soit une ⟨suite 01.⟩
AUGMENTER (V1, V2)	V1:=V1+V2, V1 et V2 ⟨entiers⟩
DIMINUER (V1, V2)	V1:=V1-V2, V1 et V2 ⟨entiers⟩
b. Opérations "booléennes"	
UNION (V1, V2, V3)	V1:=V2 ∪ V3, V1, V2 et V3 étant des ⟨suites 01.⟩ et l'union se réalisant bit par bit ; .∪1 = 1, .∪0 = . et .∪. = .

INTERSECTION (V1, V2, V3)

V1:=V2 $\cap$ V3 ; analogue à l'union mais avec . $\cap$ 1 = . , . $\cap$ 0 = 0 et . $\cap$. = .

c. Opérations aller à

SAUT (E)

allera E, E étant une étiquette qui définit un point de programme ;

SAUT SI (t, V1, V2, E)

Le saut vers l'étiquette E s'effectue lorsque V1 et V2 vérifient la relation définie par t selon la table suivante :

t	opération
1	=
2	$\neq$
3	$>$
4	$\geqslant$
5	$<$
6	$\leqslant$

Les variables V1 et V2 doivent être des ⟨entiers⟩ lorsque t $\geqslant$ 3. Elles peuvent être des ⟨suites 01.⟩ pour t $\leqslant$ 2.

d. Opérations de lecture

LIRE

Lit un symbole, c'est-à-dire affecte les valeurs correspondant à ce symbole à SYMBOLE LU et à CLASSE. Remarquons que cette lecture est automatiquement déclenchée quand on rencontre le point virgule qui indique la fin d'une action.

LIRE JUSQUA (S)	Lit et introduit dans BUFFER tous les symboles jusqu'à en trouver un de code S, celui-ci restant en SYMBOLE LU et sa classe en CLASSE.
INHIBER	Interdit la lecture du symbole suivant lors du prochain appel de LIRE (que cet appel soit automatiquement déclenché par la fin d'une action ou demandé par l'utilisateur).

e. Opérations d'édition

EDITER (V)	"ie" est augmenté de 1, et la <suite 01.> représentée par V est rangée comme dernier élément de la chaîne codée.
REEDITER (V)	Sans augmenter "ie", on range la <suite 01.> représentée par V au même endroit que le dernier élément de la chaîne codée.
SUREDITER (V1, V2)	V1 étant un <entier> et V2 une <suite 01.>, V2 est rangé dans le mot de la chaîne codée indexé par V1.

f. Opérations sur les magasins et tables

PREPARER (m,t)	PREPARER est une des opérations que l'on peut utiliser lors de la phase PREPARATION. Elle a pour but de préparer m magasins et t tables, utilisés dans la phase ACTION.

INITIALISER MAGASIN (m, LC)	Cette opération, utilisée dans la phase PREPARATION, a pour but de ranger dans le m-ième magasin, au niveau 1, une liste LC de chaînes de caractères alpha-numériques séparés par des blancs, chacune d'elles étant suivie d'une <suite 01.> .
EMMAGASINER BUFFER (m, n, c)	m et n sont des <entiers> ; c est une <suite 01.> ; cette opération range le contenu du BUFFER dans le m-ième magasin au niveau n ; c est rangé dans le magasin à la suite du contenu du BUFFER.
OUVRIR NIVEAU (m)	Un nouveau niveau est créé dans le m-ième magasin.
EFFACER NIVEAU (m)	Le dernier niveau du m-ième magasin est effacé.
SI BUFFER EN MAGASIN (m, n, B, C)	m, n et B sont des <entiers> et C est une <suite 01.> ; si la chaîne de caractères contenue dans le BUFFER existe déjà dans le n-ième niveau du m-ième magasin, la variable B prend la valeur 1 ; sinon elle prend la valeur 0. Dans le cas où B=1 le code affecté à C est celui qui suit la chaîne rangée dans le magasin, c'est-à-dire que c'est le code qui lui avait été affecté lors d'une des opérations INITIALISER MAGASIN ou EMMAGASINER BUFFER.

Remarquons que certaines des opérations décrites ci-dessus ont été spécialement conçues pour permettre le traitement des variables dans les langages ayant une structure de blocs analogue à celle d'Algol. Il est probable que l'on éprouvera le besoin d'ajouter de nouvelles opérations au fur et à mesure que l'on écrira des éditeurs pour de nouveaux langages.

Un message d'erreur étiqueté par E <entier sans signe> est formé d'une chaîne qui décrit l'erreur en question et d'une étiquette ; cette dernière indique le point du programme où continuer (ou terminer) l'édition.

III.3. Exemple d'utilisation

Revenons à l'édition des chaînes données au paragraphe III.1. On veut reconnaître les suites de symboles de la forme A B^m C^n P et produire la chaîne codée où les codes de A, B, C et P sont 0010, 0100, 0101 et 1000.

La table d'états et les actions correspondantes sont résumées dans le tableau ci-dessous. Observons qu'il a été nécessaire d'ajouter de nouveaux états à ceux de la page 26 pour réaliser la séquence d'actions voulue. Signalons que dans cet exemple les codes de B ou de C ne figureront pas dans la chaîne codée lorsque la valeur correspondante de m ou de n sera égale à zéro.

ACTIONS (Commentaires)	ETATS	A	B	C	P	(Commentaires)
	0	1	e	e	e	
éditer A ; m:=n:=0 ;	1	e	2	6	7	après A
éditer B ; augmenter m ;	2	e	3	4	8	après AB
augmenter m ;	3	e	3	4	8	après ABB...
éditer m ; éditer C ; augmenter n ;	4	e	e	5	9	après AB...C
augmenter n ;	5	e	e	5	9	après AB...C... ou AC...
éditer C ; augmenter n ;	6	e	e	5	9	après AC
éditer P ;	7	e	e	e	e	après AP (final)
éditer m ; éditer P ;	8	e	e	e	e	après AB...P (final)
éditer n ; éditer P ;	9	e	e	e	e	après AB...C...P ou AC...P (final)

Programme en langage d'édition

```
CLASSES :

LETTRE A (A) ;
LETTRE B (B) ;
LETTRE C (C) ;
LETTRE P (P) *

ETATS :  1, 10, 10, 10 ; c 0, état initial ;
        10,  2,  6,  7 ; c 1, après A ;
        10,  3,  4,  8 ; c 2, après AB ;
        10,  3,  4,  8 ; c 3, après ABB... ;
        10, 10,  5,  9 ; c 4, après AB...C ;
        10, 10,  5,  9 ; c 5, après A....C... ;
        10, 10,  5,  9 ; c 6, après AC ;
        10, 10, 10, 10 ; c 7, après AP, final ;
        10, 10, 10, 10 ; c 8, après AB...P, final ;
        10, 10, 10, 10 ; c 9, après A...C...P, final ;
        10, 10, 10, 10 ; c 10, erreur    *

PREPARATION :

AFFECTATION (A, b 0010), AFFECTATION (B, b 0100),
AFFECTATION (C, b 0101), AFFECTATION (P, b 1000) *

ACTIONS :

A1 : EDITER (A), AFFECTATION (m, 0), AFFECTATION (n, 0) ;
A2 : EDITER (B) ; AUGMENTER (m, 1) ;
A3 : AUGMENTER (m, 1) ;
A4 : EDITER (m), EDITER (C), AUGMENTER (n, 1) ;
A5 : AUGMENTER (n, 1) ;
A6 : EDITER (C), AUGMENTER (n, 1) ;
A7 : EDITER (P), SAUT (FIN) ;
A8 : EDITER (m), EDITER (P), SAUT (FIN) ;
A9 : EDITER (n), EDITER (P), SAUT (FIN) ;
A10 : SAUT (E1) *

ERREURS :
E1 : ERREUR NO 1, FIN *
```

III.4. Réalisation pratique

Dans ce paragraphe nous donnerons quelques indications relatives à la construction du compilateur du langage d'édition. Deux principes la gouvernent :

1. Le langage défini par l'utilisateur sera exploité beaucoup plus souvent que le langage d'édition.

2. L'utilisateur doit être un programmeur expérimenté. En conséquence il est normal que les messages d'erreurs qui lui sont fournis à l'intérieur du langage d'édition soient moins perfectionnés que ceux qu'il aura la possibilité de fournir à son client. Néanmoins, l'intérêt d'un compilateur étant lié au nombre d'erreurs qu'il peut détecter, les messages destinés à l'utilisateur pour la mise au point de son éditeur sont indispensables.

Compte tenu de ce principe et du fait que nous avons à notre disposition un moyen simple et puissant d'analyse d'un langage, il est élégant et séduisant d'employer le mécanisme des tables d'états dans la construction du compilateur lui-même. Cela est d'autant plus intéressant que la syntaxe que nous avons décrite peut, grâce à l'absence d'imbrications, être ramenée à une grammaire d'états finis.

Selon le premier principe il n'y a aucun inconvénient à écrire en Algol la partie du compilateur qui construit la table

d'états, et qui range les paramètres nécessaires à l'exploitation du programme écrit par le client de l'utilisateur. En conséquence, pour que cette exploitation soit efficace, seul est nécessaire un noyau extrêmement performant, écrit en langage machine.

Compte tenu de la taille considérable des tables d'états dont l'utilisateur peut avoir besoin, il est nécessaire d'utiliser la mémoire d'une façon économique, par exemple, en rangeant deux états par mot-machine. Dans le cas de la machine IBM 7044 pour laquelle le noyau sera construit, l'accès aux parties adresse et décrément entraîne une perte de temps, mais qui sera minime grâce à l'utilisation des registres d'index. Il est donc souhaitable de prévoir quelques procédures en code, qui pourront être incorporées au programme Algol de génération de tables. Ces procédures permettront le rangement des entiers en partie adresse ou décrément d'un mot.

Le noyau (3) du paragraphe III.1, écrit en langage machine pour l'exploitation, se chargera d'extraire le nouvel état, d'une partie adresse ou décrément selon le cas.

III.4.1. Gestion de tables

La gestion des tables définies par l'utilisateur pose elle aussi un problème important. En effet il doit être possible de définir un nombre arbitraire de magasins et de tables sans connaître a priori

la quantité d'information qu'ils doivent recevoir. C'est dans ce but que J. GARWICK [Gar 2] a proposé un procédé de gestion de tables qui les réorganise en mémoire lorsque l'une d'entre elles déborde. L'espace disponible est initialement partagé de façon équitable entre les m+t magasins et tables indiqués par l'opération d'initialisation PREPARER (m, t). Dans ce qui suit nous groupons les magasins et les tables sous le nom commun "table". Tant qu'aucune des tables ne dépasse pas l'espace qui lui est attribué, le rangement dans chacune d'elles se fait normalement. Mais si l'une des tables est pleine et que l'on désire lui ajouter un élément, on procède à une réorganisation pour redistribuer l'espace total disponible de chaque table. Le programme détaillé de cette réorganisation est donné en [Gar 2].

Enfin, les rangements sont effectués dans les magasins, selon la technique des tables binaires [C1]. Cette méthode évite toute limitation sur la longueur des identificateurs, permet un gain de place et diminue le temps de recherche. On peut la mettre en oeuvre en plaçant le symbole en partie décrément et le pointeur vers un alternant éventuel en partie adresse.

III.4.2. Actions

Nous avons remarqué en III.1 que l'un des buts à poursuivre était d'obtenir un langage d'édition aisément compilable en un

programme efficace. La reconnaissance des classes et la mise en table des états indiqués par l'utilisateur ne constituent pas un problème à la compilation. En revanche, seul un choix judicieux des opérations et de leur structure d'appel permet de rendre performante l'édition des programmes du client de l'utilisateur. Les opérations décrites au § III.2.2 ont la structure de macro-instructions qui peuvent être aisément assemblées par un macro-assembleur et qui produisent un programme efficace.

III.4.3. Remarques sur le langage d'édition

L'exemple traité au § III.3 ne donne qu'une idée très modeste des possibilités du langage d'édition. Nous avons déjà utilisé ce langage pour la partie d'édition d'un compilateur Algol simplifié ne traitant que les variables du type entier. Il a aussi été utilisé expérimentalement pour réaliser la partie d'édition d'un compilateur COBOL.

L'écriture du compilateur du langage d'édition est actuellement dans sa phase finale. Il a pu être décrit dans son propre langage [Jor] ; ceci présente l'avantage de rendre sa description indépendante de la machine.

Dans ce qui précède nous avons essayé de définir un langage dont la simplicité permette la génération d'un programme d'exploitation efficace. Les deux améliorations que nous suggérons ci-dessous

rendront le langage d'édition plus complexe, et le gain apporté par leur réalisation pratique dépendra en grande partie de la facilité avec laquelle les utilisateurs se serviront de ce compilateur. Ce sont :

a) Possibilité d'imbrication. Le langage d'édition deviendrait beaucoup plus puissant si l'on pouvait introduire de nouvelles classes, de nouveaux états et de nouvelles actions à un niveau quelconque. La structure du langage serait alors comparable à celle des blocs en Algol.

b) Impression du programme édité. Il serait intéressant d'introduire plusieurs opérations pour l'impression du programme édité. Par exemple pour un langage tel qu'Algol ces opérations permettraient à l'utilisateur de fournir à son client des impressions supplémentaires repérant les 'début' et 'fin' selon leur niveau d'imbrication, de donner des messages d'erreurs plus faciles à interpréter, etc.

III.5. Transformation des règles sous FNB en tables d'états *

On propose ici un algorithme de transformation des grammaires "context-free" en tables d'états, c'est-à-dire en tables qui définissent un automate d'états finis équivalent.

Le problème de savoir si un langage "context-free" (C.F.) peut être analysé par un automate d'états finis étant indécidable [BPS, La], nous n'assurons pas dans tous les cas le succès de l'algorithme, quoique celui-ci puisse être utile pour la construction de compilateurs et la création de langages. En effet, la simulation d'un automate d'états finis sur un calculateur digital est d'ordinaire plus efficace que la simulation d'un automate à pile, nécessaire pour analyser les langages C.F. généraux.

Le principe de base de l'algorithme est très simple : étant donnée une grammaire C.F., on sait la transformer en une grammaire équivalente dont les règles sont de forme standard [Gre 1, Gre 2, Gre 3], c'est-à-dire toutes de la forme :

$$A \longrightarrow a\varphi$$

où "A" appartient au vocabulaire non-terminal V_N et "a" au vocabulaire terminal V_T ; φ est une chaîne sur V_N, ou bien la chaîne vide.**

(*) Les notations utilisées dans ce paragraphe sont celles du chapitre IV.

(**) En réalité nous utilisons ici une extension de la forme standard ; φ peut être formé d'éléments appartenant à $V_T \cup V_N$, ce qui simplifie l'algorithme de transformation.

Si, pour toutes les règles de la grammaire, φ est soit un symbole non-terminal, soit le symbole vide λ, cette grammaire est alors d'états finis. Si la longueur de φ est supérieure à un, alors :

1) On crée un nouveau non-terminal A' et on remplace la règle ci-dessus par :

$$A \longrightarrow a\ A' \quad \text{et} \quad A' \longrightarrow \varphi \ ;$$

s'il existe déjà une règle $A'' \longrightarrow \varphi$ résultant d'une application précédente de (1), alors on utilise le non-terminal A" au lieu de A'.

2) On essaie une nouvelle transformation en forme standard.

Si l'algorithme se termine, l'automate d'états finis équivalent peut très bien être non déterministe ou non minimal. Cependant, les méthodes classiques permettent, si on le désire, sa transformation en un autre automate déterministe et minimal [Gi 1, Har].

Remarquons que même si l'algorithme, appliqué à une grammaire G donnée, ne converge pas, il peut exister une grammaire d'états finis équivalente à G. Par exemple, le langage défini par $A \longrightarrow \varkappa\ A\ \varkappa$ et $A \longrightarrow \varkappa$ peut être représenté par une grammaire d'états finis bien que l'algorithme échoue dans ce cas.

III.5.1. Exemple

La métavariable ⟨nombre⟩ est définie en Algol de la façon suivante :

N ⟶ U | sU

U ⟶ D | E | DE

D ⟶ J | F | JF

E ⟶ eI

F ⟶ pJ

I ⟶ J | sJ

J ⟶ dJ | d

où : N est un ⟨nombre⟩, U un ⟨nombre sans signe⟩, D un ⟨nombre décimal⟩, E une ⟨partie exposant⟩, F une ⟨fraction⟩, I un ⟨entier⟩, J un ⟨entier sans signe⟩, s un "±", p un ".", e un "10", d un ⟨chiffre⟩ et où $N,U,D,E,F,I,J \in V_N$ et $s,p,e,d \in V_T$. La transformation réitérée en forme standard conduit aux règles suivantes :

N ⟶ d | dJ | pJ | dF | dP | eI | dE | dQ | pQ | dT | dR | sU

U ⟶ d | dJ | pJ | dF | dP | eI | dE | dQ | pQ | dT | dR

D ⟶ d | dJ | pJ | dF | dP

E ⟶ eI

F ⟶ pJ

I ⟶ d | dJ | sJ

J ⟶ d | dJ

P ⟶ dF | dP

Q ⟶ dE | dQ

R ⟶ dT | dR

T ⟶ pQ

où l'on introduit les nouveaux non-terminaux P,Q,R et T.

Les règles ci-dessus correspondent à l'automate d'états finis non déterministe dont on trouvera la table d'états en Table 1. On

accomplit la transformation de cette table dans une table d'états déterministe comme en [Gi 1, Har], en appelant :

S_1 l'état initial équivalent à N
S_2 l'état U
S_3 l'union des états λ, J, F, P, E, Q, T, R
S_4 l'union des états J, Q
S_5 l'union des états λ, J, E, Q
S_6 l'état I
S_7 l'état J
S_8 l'union des états λ, J (état final)

Cela produit comme résultat final la table donnée en Table 2. Cette dernière coïncide avec celle donnée en [Na 3] pour le premier passage d'un compilateur Algol.

	s	d	p	e
N	U	λ,J,F,P,E,Q,T,R	J,Q	I
U		λ,J,F,P,E,Q,T,R	J,Q	I
D		λ,J,F,P	J	
E				I
I	J	λ,J		
F			J	
J		λ,J		
P		F,P		
Q		E,Q		
R		T,R		
T			Q	

Table 1

	s	s	p	e
S_1	S_2	S_3	S_4	S_6
S_2	e	S_3	S_4	S_6
S_3	e	S_3	S_4	S_6
S_4	e	S_5	e	e
S_5	e	S_5	e	S_6
S_6	S_7	S_8	e	e
S_7	e	S_8	e	e
S_8	e	S_8	e	e

e - erreur

Table 2

III.5.2. Remarques

On pourra facilement programmer cet algorithme sur un calculateur digital.* Nous avons déjà un programme qui transforme des règles C.F. en règles de forme standard [Ndx]. Ce programme ne traite pas les règles récursives à gauche mais on pourra facilement y incorporer la méthode décrite en [Ks]. C'est d'ailleurs un des avantages de la démarche que nous proposons : les non terminaux introduits par la méthode en question sont automatiquement éliminés lors du calcul de l'automate déterministe minimal. Il serait intéressant d'utiliser l'algorithme proposé en mode conversationnel, comme dans [Ev] : l'utilisateur pourrait essayer de déterminer les parties d'un langage analysables par un automate d'états finis. Même si certaines parties d'un langage ne peuvent être traitées que par un automate à pile, il serait intéressant d'essayer d'utiliser plusieurs passages, quelques-unes reconnaissant les chaînes d'états finis.

Sujets de recherche

1. Réalisation du programme décrit dans ce paragraphe. Comparaison avec l'algorithme proposé par CHOMSKY [Ch 2] et un autre, récemment proposé en [Joh].

(*) L'inclusion de Lisp en Algol décrite au chapitre V est d'ailleurs le moyen plus naturel pour réaliser cet algorithme en machine.

2. Détermination des restrictions que l'on doit apporter aux langages C.F. pour pouvoir automatiser encore plus la génération de tables d'états. Si l'on pouvait séparer les cas d'auto-imbrications, il nous semble qu'un analyseur d'états finis pourrait reconnaître de grandes parties des langages du type d'Algol.

IV. Analyse

"Les grammaires sont pour les auteurs
ce qu'un luthier est pour un musicien."
(Voltaire, Pièces posthumes)

"Purus grammaticus, purus asinus."
(Erasme)

Nous traitons dans ce chapitre les problèmes relatifs au noyau du compilateur que nous voulons construire : l'analyseur. Etant donnée l'abondance des publications dans le domaine de l'analyse syntaxique des langages(*), nous avons jugé utile de présenter d'abord un bref résumé des travaux théoriques existants, suivi d'une classification des différentes méthodes d'analyse déjà employées en compilation ou en traduction automatique. Dans cette partie l'accent est mis sur l'établissement d'un lien entre la théorie et la pratique.

Nous étudions ensuite un sujet auquel on a prêté fort peu d'attention jusqu'à présent : comment se servir du résultat d'une analyse. Dans ce but nous proposons une voie qui permet à l'utilisateur de

---

(*) Une bibliographie assez complète se trouve en [F1 5].

préciser la notion d'opérateur et d'opérande au moyen de regroupements des éléments de la chaîne donnée. Les règles de grammaire sous FNB et les indications fournies par l'utilisateur pour réaliser ces regroupements forment le langage d'analyse décrit à la fin de ce chapitre.

IV.1. Analyse syntaxique

IV.1.1. Aspects théoriques

L'une des plus séduisantes parmi les théories mathématiques des langages a été proposée par N. CHOMSKY en 1957 [Ch 1]. Depuis cette époque un nombre croissant de mathématiciens, linguistes et programmeurs s'intéressent à cette théorie. L'un des principaux mérites de CHOMSKY est d'avoir su définir de manière concise et élégante un ensemble infini de chaînes - le langage - à partir d'un nombre fini d'éléments - la grammaire.

Nous ne répétons pas ici les définitions de base de la théorie de CHOMSKY. On trouvera en [Ch 2, Gi 2, HSS, Va, Gro] présentés de façon claire, les principaux résultats de cette théorie. Les notations utilisées ici sont les suivantes :

- V_T : vocabulaire terminal ou ensemble de symboles de base ; chaque élément est représenté par une lettre minuscule ;
- V_N : vocabulaire non terminal ou ensemble de métavariables ; l'élément est représenté par une lettre majuscule ;
- S : élément spécial de V_N, appelé axiome ou programme ;
- R : ensemble de règles de réécriture du type

$$\varphi \longrightarrow \psi$$

où φ et ψ sont des chaînes sur $V_T \cup V_N$. CHOMSKY a distingué des classes de langages selon le type de règles R et a donné les mécanismes (automates) les plus simples capables de les analyser.

Jusqu'à présent les mathématiciens se sont intéressés à la détermination des propriétés booléenes et d'inclusion des différentes classes de langages chomskiens. Les résultats obtenus, dont on trouvera en [La] un résumé intéressant, permettent presque toujours soit de fournir un algorithme qui réalise l'opération voulue, soit de montrer que cet algorithme n'existe pas (le problème est indécidable).

Les problèmes de décision résolus par les mathématiciens concernent actuellement des sous-ensembles des langages de CHOMSKY (linéaires, minimaux, déterministes et même sous-ensembles des langages d'états finis). Ces résultats n'apportent malheureusement pas beaucoup d'aide aux programmeurs. Bien que l'indécidabilité d'un problème nous avertisse qu'il est absurde d'essayer de le programmer en machine, cet avertissement n'a pour l'utilisateur qu'une portée limitée (penser à l'impossibilité du mouvement perpétuel, qui n'a pas empêché des chimistes sérieux de chercher à réduire les effets de la friction). Inversement, même si un problème est décidable, il est impossible de savoir s'il demande une puissance de calcul (taille de mémoire et temps d'exécution) inconnue à ce jour. Quelques mathématiciens essaient actuellement de définir des automates à mémoire finie où l'on attache un coût à chaque règle ; les problèmes de décision portent sur l'existence d'un algorithme défini sur ces automates, en précisant la taille de mémoire disponible et le coût total d'exécution. Ceci constitue déjà un rapprochement entre la théorie et la pratique.

En ce qui concerne la compilation, on peut dire que l'intérêt manifesté par les programmeurs envers la théorie des langages a commencé lorsque l'on a montré que la définition syntaxique des langages de programmation proposée par BACKUS [Bac] était équivalente à celle du type 2 de CHOMSKY, que l'on appelle "context-free" (dans ce cas les φ des règles R se réduisent à un seul élément de V_N que nous appelons sujet de la règle). Le succès de la FNB est dû au fait qu'elle permet de définir de grandes parties d'un langage avec un nombre raisonable de règles. Le sens des méta-variables est donné dans la partie sémantique du langage ; on y fait souvent sur la syntaxe de nouvelles remarques qui seraient impossibles ou lourdes à exprimer sous FNB. Ainsi pour définir un identificateur ayant au maximum six lettres, il est commode de définir d'abord une partie syntaxique

<identificateur> ::= <lettre> |<lettre> <identificateur>

et de la faire suivre d'une partie sémantique où l'on indique qu'un <identificateur> ne peut contenir que six lettres au maximum. On pourrait bien sûr définir l'identificateur en question par une partie purement syntaxique ayant un nombre élevé de règles sous FNB mais cela n'a aucun intérêt pratique. Le fait d'avoir un nombre raisonnable de règles de réécriture est primordial pour définir de façon pratique un langage C.F.

Parmi les résultats apportés par la théorie mathématique des langages, deux nous intéressent particulièrement. Le premier nous

dit que la détermination automatique de l'ambiguité des langages C.F. est impossible : seule l'expérience sur un langage particulier peut nous garantir qu'il n'est pas ambigu. Si cette détermination avait été possible, on aurait pu éviter certaines des ambiguités embarassantes du premier rapport Algol. Ce résultat négatif a amené les théoriciens à déterminer des sous-ensembles des langages C.F. pour lesquels l'on peut assurer qu'il n'y a pas d'ambiguités. Les langages déterministes définis par GINSBURG [Gi 2] en sont un exemple. Les langages définis par FLOYD [F1 3, 4] et KNUTH [Kn] en sont un autre. A ce propos, remarquons que FLOYD et KNUTH utilisent dans leurs analyseurs des piles dont le fonctionnement s'apparente à celui des piles utilisées dans les compilateurs réels. Les mathématiciens, intéressés par les propriétés intrinsèques aux langages, utilisent une pile beaucoup plus simple qui possède miraculeusement le don d'ubiquité de Sabine* : on crée de nouvelles piles pendant l'analyse au fur et à mesure que l'on en a besoin. La simulation de ces piles par un calculateur séquentiel est lente ; elle sera décrite au § IV.2.1.

Le dernier résultat d'intérêt pratique apporté par la théorie des langages est lui aussi négatif : il est impossible de déterminer si un langage C.F. ne peut pas aussi être décrit par un langage

---

(*) "Elle pouvait à son gré se multiplier et se trouver en même temps de corps et d'esprit, en autant de lieux qu'il lui plaisait souhaiter." (Marcel Aymé)

d'états finis. Ce résultat a d'ailleurs déjà été mentionné au § III.5 où nous avons proposé un algorithme qui permet dans certains cas de transformer une grammaire C.F. en un autre grammaire d'états finis qui lui est équivalente.

Dans ce qui suit nous présentons deux classifications des méthodes d'analyse des langages C.F. La première, dûe à GRIFFITHS et PETRICK [GP], a eu pour but de vérifier l'efficacité relative de ces méthodes. La seconde constitue une approche plus intuitive du problème.

IV.1.2. Classification de GRIFFITHS et PETRICK

Afin d'en comparer l'efficacité, GRIFFITHS et PETRICK ont exprimés certains algorithmes d'analyses des langages C.F. sous la forme d'ensembles de règles (productions) pour une même machine abstraite. Ils ont ensuite simulé sur calculateur le fonctionnement de cette machine abstraite.

La machine abstraite choisie par GRIFFITHS et PETRICK a la puissance d'une machine de Turing. Elle est décrite par le triplet

$$\{\alpha, \beta, r\}$$

où α et β sont des rubans fonctionnant comme des piles et r est un ensemble de productions du type :

$$(x_\alpha , x_\beta) \longrightarrow (x'_\alpha , x'_\beta)$$

Une production est dite applicable lorsque les sommets de α et β sont respectivement identiques aux chaînes x_α et x_β. L'effet de cette application est de remplacer la partie commune de α et x_α par x'_α et celle de β et x_β par x'_β.

Dans une version déterministe de cette machine, on ordonne les règles et on recherche séquentiellement l'applicabilité. Lorsqu'une règle est applicable, on réécrit α et β comme plus haut, et le balayage des règles recommence au début. La machine s'arrête si aucune règle n'est applicable.

Dans une version non déterministe on essaie en parallèle les règles applicables, la condition d'arrêt étant analogue à celle de la machine déterministe. Nous présentons au § V.5.1 un programme de simulation de cette machine.

Les algorithmes étudiés par GRIFFITHS et PETRICK sont :

		Abréviation utilisée
a.	analyse descendante non sélective	(NTB)
b.	analyse descendante sélective	(STB)
c.	analyse ascendante non sélective	(NBT)
d.	analyse ascendante sélective	(SBT)
e.	analyse par substitution directe non sélective	(NDS)
f.	analyse par substitution directe sélective	(SDS)

L'analyse est descendante ou ascendante selon la direction ascendante ou descendante adoptée pour créer l'arbre d'analyse. Nous reviendrons sur cette définition au paragraphe suivant.

Les moyens utilisés pour réduire la longueur des chemins parcourus inutilement caractérisent la sélectivité d'un algorithme : quand on simule séquentiellement la machine non déterministe, ces chemins correspondent aux retours en arrière nécessaires pour tenter de suivre une deuxième séquence de calcul lorsque la première conduit à un blocage.

L'évaluation du degré de sélectivité d'un algorithme d'analyse réel est difficile ; une comparaison entre la sélectivité de deux algorithmes est cependant possible si l'on peut les exprimer sous forme de règles de la machine abstraite. Cette transformation de l'algorithme peut malheureusement fausser son véritable fonctionnement, ce qui constitue une critique essentielle faite aux travaux de GRIFFITHS et PETRICK.

L'algorithme par substitution directe s'apparente à celui de l'analyse ascendante : il consiste d'abord à remplacer, de gauche à droite sur la chaîne donnée, les éléments terminaux par les éléments de V_N indiqués par les règles terminales* ; on utilise ensuite les autres règles pour remplacer successivement les éléments de $V_T \cup V_N$ par d'autres éléments non terminaux s'approchant de plus en plus de l'axiome.

(*) C'est-à-dire les règles du type $A \rightarrow a$, $A \in V_N$, $a \in V_T$.

L'analyse ascendante utilise un marqueur pour distinguer les éléments de la chaîne donnée pour lesquels un sous-arbre a déjà été construit : on essaie alors d'appliquer une règle utilisant les éléments à droite du marqueur, de façon à complèter ce sous-arbre.

La simulation en calculateur de la machine abstraite de GRIFFITHS et PETRICK et les essais de plusieurs grammaires et chaînes données ont montré que l'algorithme d'analyse ascendante sélective était bien plus efficace que celui d'analyse descendante sélective. D'autre part, plus un algorithme est sélectif, plus on peut en attendre d'efficacité. Nous présentons au § IV.2.4 une version en Algol d'un algorithme ascendant sélectif que nous recommandons pour la réalisation pratique du langage d'analyse décrit au § IV.5.

L'analyse que nous appelons ascendante non sélective (§ IV.2.3) s'apparente plutôt au NDS de GRIFFITHS et PETRICK. La méthode de KUNO-OETTINGER décrite au § IV.2.2 est, selon GRIFFITHS et PETRICK, à la fois STB et NTB pourvu que la grammaire soit déjà donnée sous forme standard. L'algorithme présenté aux paragraphes IV.2.1 et V.5.1 est calqué directement sur l'analyseur NTB de GRIFFITHS et PETRICK.

IV.1.3. Classification intuitive

Nous classons ici certains des algorithmes existants en essayant de les séparer selon les caractéristiques suivantes :

		abréviation
a. Direction	descendante	DSC
	ascendante	ASC
b. Sélectivité	sélective	SEL
	non sélective	NSL
c. Nombre d'analyses	multiple	MUL
	unique ou simple	SPL
d. Généralité	général	GEN
	spécial (déterministe)	SDT
	(non déterministe)	SND

Avant d'exposer la classification proprement dite, précisons ce que nous entendons par chacun de ces critères.

a. Direction

Précisons d'abord le problème posé au calculateur. Celui-ci possède en mémoire un tableau, contenant les règles de la grammaire ; il doit retrouver une dérivation qui, du symbole S, conduit à la chaîne donnée. Au départ, son but est d'obtenir S, sommet de l'arbre par lequel nous représentons cette structure. Mais ce but ne peut être atteint directement. Les noeuds de l'arbre deviendront des

buts* intermédiaires, qui seront placés dans une pile. Au moment de l'analyse du i-ème symbole de la chaîne, cette pile contient un certain nombre d'éléments qui correspondent aux buts cherchés en partant des i premiers symboles. Le but qui se trouve au sommet de la pile est éliminé quand il a pu être obtenu à partir des éléments de la chaîne déjà lus.

L'ordre dans lequel ces buts intermédiaires sont placés sur la pile dépend du type d'analyse effectué. Si l'analyse est ascendante, on place d'abord dans la pile les buts correspondants aux noeuds de l'arbre les plus proches des symboles de la chaîne donnée. Quand l'analyse est descendante, les premiers buts placés dans la pile correspondent aux noeuds les plus proches de S.

Nous pouvons exprimer cette différence d'une autre façon : si la grammaire est présentée avec le formalisme proposé par TAYLOR, TURNER et WAYCHOFF et utilisé pour la carte syntaxique d'ALGOL [TTW], l'analyse descendante conduit à examiner (c'est-à-dire, essayer comme but) les métavariables en allant de haut en bas dans la carte. Dans l'analyse ascendante, l'examen des métavariables est fait de bas en haut.

(*) Plus précisément, un "but" est un élément du vocabulaire non terminal (métavariable) que l'on essaye d'obtenir et qui sera un noeud de l'arbre si cette tentative réussit.

b. Sélectivité

Nous étendons ici la notion de sélectivité donnée au § IV.1.2.* Nous appelons sélectivité la quantité de prétraitements que l'on fait subir à une grammaire afin de réduire le nombre d'essais dans la phase d'analyse proprement dite. La quantité d'information extraite par le prétraitement peut se présenter soit sous forme de tables, soit sous forme de modifications des règles de grammaire.

c. Nombre d'analyses

Un algorithme d'analyse est dit multiple quand il détermine tous les arbres associés à une chaîne donnée. Lorsque l'algorithme s'arrête après avoir réalisé la première analyse, celle-ci est dite unique ou simple.

L'analyse multiple ne présente évidemment d'intérêt pratique que dans le cas des langages ambigus. Les analyseurs multiples sont surtout utilisés pour l'analyse des langages naturels. Il est en général facile de transformer un analyseur simple en un analyseur multiple, et inversement.

d. Généralité

Un algorithme est général quand il s'applique à n'importe quelle grammaire C.F. Les algorithmes spéciaux s'appliquent le plus

---

(*) Précisons à nouveau que la sélectivité d'un algorithme est une propriété difficilement mesurable.

souvent à des grammaires sous forme spéciale. En ce qui concerne la compilation, il est intéressant d'avoir des grammaires spéciales à la fois pour supprimer les ambiguités et pour obtenir un analyseur rapide ; dans ce cas les opérations sur la pile sont plus complexes que celles qu'utilisent les théoriciens des langages. Ces grammaires spéciales sont déterministes.

Passons à la classification de certains algorithmes existants, selon les caractéristiques définies ci-dessus. Le compilateur de BROOKER et MORRIS [BM, BMMR, Mor] utilise un analyseur DSC, NSL, SPL et GEN. En réalité, leur analyseur n'est pas tout à fait général : il ne peut pas traiter les règles récursives à gauche. L'algorithme de CHEATHAM [CL, CS] appartient à cette même classe ; il peut cependant traiter quelques cas de récursivité à gauche.

L'algorithme d'IRONS [Ir 1, Ir 3] et celui décrit au § IV.2.4 sont ASC, SEL, SPL et GEN. L'algorithme de KUNO et OETTINGER, décrit au § IV.2.2, est DSC, SEL, MUL et SND. Celui décrit au § IV.2.3 est ASC, NSL, MUL et SND.

Nous pouvons dire que la sélectivité d'un algorithme spécial déterministe est totale, puisqu'il n'a pas à effectuer de retours en arrière. De plus, l'analyse y est évidemment toujours unique. Ainsi, les algorithmes de FLOYD [F1 3, F1 4] et KNUTH [Kn] sont ASC, SDT.

IV.2. Description de quelques méthodes d'analyse

Parmi les méthodes d'analyse mentionnées dans les paragraphes précédents, nous en avons choisi quatre qui peuvent être considérées comme des prototypes d'analyseurs existants. Deux d'entre eux représentent l'analyse descendante, non sélective ou sélective, et les deux autres les versions correspondantes des analyseurs ascendants. La description de ces quatre algorithmes montre bien les différents aspects, d'ailleurs difficilement mesurables, de la sélectivité.

En ce qui concerne leur réalisation pratique, dans le cadre d'un compilateur dirigé par la syntaxe, seule l'analyse ascendante sélective paraît appropriée. Seul l'algorithme d'analyse ascendante sélective n'effectue pas d'analyse multiple ; il pourrait facilement le faire moyennant de petites modifications. Il est également facile de faire des modifications aux trois premiers afin de ne réaliser qu'une seule analyse.

IV.2.1. Analyse descendante non sélective

Nous considérons ici l'algorithme d'analyse descendante non sélective proposé par GRIFFITHS et PETRICK. Il consiste à simuler leur machine à deux piles, dont les règles sont déduites de la grammaire du langage donné. La présentation du programme de simulation de cet automate sera faite au § V.5.1, après la description

du langage de listes qu'il utilise. Ce programme emploie la récursivité pour réaliser séquentiellement l'automate non déterministe.

Dans ce qui suit nous nous limitons à décrire la transformation des règles de la grammaire dans les règles de l'automate : pour chaque règle de la grammaire du type

$$U \longrightarrow \varphi$$

où φ est une chaîne sur $V_T \cup V_N$, on obtient la règle suivante :

$$(\Lambda, U) \longrightarrow (\Lambda, \varphi) \qquad (1)$$

où Λ est la chaîne vide ; une règle de ce type s'applique pour n'importe quelle configuration de la pile α qui contient la chaîne à analyser. Pour chaque élément "a" de V_T on écrit aussi la règle de l'automate :

$$(a, a) \longrightarrow (\Lambda, \Lambda) \qquad (2)$$

Cette règle sert à "comparer" un symbole de la chaîne à analyser avec le symbole qui a été mis au sommet de la pile β par une règle du premier type. L'algorithme fonctionne par des "essais de génération" : l'axiome est mis au sommet de la pile β, précédé d'une marque $\#$, $\# \notin V_T \cup V_N$; cette marque est aussi utilisée comme dernier élément de la chaîne donnée rangée dans la pile α. Les règles du type (1) sont essayées l'une après l'autre ; parmi ces règles, une au moins doit être applicable. L'application de l'une d'elles équivaut à remplacer la partie gauche de la règle correspondante de la grammaire par sa partie droite. Ces substitutions se poursuivent jusqu'à ce

qu'une règle du type (2) soit applicable. Celle-ci efface les deux éléments identiques du sommet de chaque pile, ce qui revient à admettre que les applications antérieures des règles du type (1) étaient valables jusqu'à ce point de l'analyse. L'analyse est terminée lorsqu'il ne reste que les marques # dans les piles α et β. L'arbre d'analyse syntaxique peut être facilement construit à partir d'une série de nombres indiquant le numéro des règles qui ont été appliquées.

Exemple : La grammaire

$$V_T = \{ a \} \quad , \qquad V_N = \{ S, A, B \}$$

S ⟶ AB
A ⟶ aA
A ⟶ a
B ⟶ aB
B ⟶ a

est décrite par les règles d'automate suivantes :

1 $(\Lambda, S) \longrightarrow (\Lambda, AB)$
2 $(\Lambda, A) \longrightarrow (\Lambda, aA)$
3 $(\Lambda, A) \longrightarrow (\Lambda, a)$
4 $(\Lambda, B) \longrightarrow (\Lambda, aB)$
5 $(\Lambda, B) \longrightarrow (\Lambda, a)$
6 $(a, a) \longrightarrow (\Lambda, \Lambda)$

Nous présentons ci-dessous les différentes configurations des piles α et β pour la première des analyses de la chaîne

a a a a #

pas	α	β	n° de la règle appliquée
1	a a a a #	S #	1
2	a a a a #	A B #	2
3	a a a a #	a A B #	6
4	a a a #	A B #	2
5	a a a #	a A B #	6
6	a a #	A B #	3
7	a a #	a B #	6
8	a #	B #	5
9	a #	a #	6
10	#	#	

Cette liste ne décrit que les applications qui ont réussi, c'est-à-dire qui ont conduit à la fin de l'analyse. Ainsi par exemple, au pas 6, la règle 2 est à nouveau applicable : elle conduit à l'élimination prématurée d'un autre "a", ce qui n'est constaté qu'après vidage de la pile α , la pile β contenant AB#. Comme aucune règle ne s'applique à cette configuration de α et β , le retour en arrière est inévitable. Cet exemple sera repris au § V.5.1 afin de préciser le fonctionnement de l'algorithme.

Remarquons pour terminer que notre simulation de cet automate dans un calculateur séquentiel interdit parfois l'utilisation de règles récursives à gauche.* Plus précisément, les règles du type (3)

(*) C'est d'ailleurs une restriction très fréquente dans les algorithmes d'analyse descendante.

ci-dessous ne sont pas permises.

$$
\begin{aligned}
&U_1 \rightarrow U_2 \quad \varphi_1 \\
&U_2 \rightarrow U_3 \quad \varphi_2 \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&U_i \rightarrow U_1 \quad \varphi_i
\end{aligned}
\qquad (3)
$$

avec les $U_i \in V_N$ et les φ_i sur $V_T \cup V_N$.

Ces règles conduisent à une boucle qui empile continuellement de nouvelles métavariables dans la pile. On peut surmonter cette difficulté au moyen d'une vérification avant application d'une production : lorsque le nombre d'éléments de la pile β (pile des métavariables) est supérieur à celui de α (chaîne d'entrée), on revient en arrière. Cette vérification, appelée "shaper", est utilisée au § V.5.1. On emploie fréquemment cet artifice dans les analyseurs des langues naturelles, la chaîne d'entrée - une phrase - étant de longueur réduite. Son utilisation dans l'analyse de tout un programme Algol serait absurde.

IV.2.2. Analyse descendante sélective

L'algorithme d'analyse de KUNO-OETTINGER décrit ici est classé comme descendant sélectif [KO, Oe, HSS]. La sélectivité consiste dans ce cas à exiger que toutes les règles de la grammaire soient sous une forme spéciale, dite forme standard. La transformation d'une grammaire

C.F. générale en une autre dont les règles sont toutes sous cette forme peut se faire automatiquement. Les travaux de GREIBACH [Gr 1, Gr 2, Gr 3] indiquent comment réaliser cette transformation ; ils montrent aussi qu'elle conserve les ambiguités éventuelles du langage. Ce dernier résultat est important pour l'analyse des langues naturelles.

Les règles d'une grammaire C.F. sont sous forme standard lorsqu'elles s'écrivent comme suit :

$$X \longrightarrow a\, Y_1\, Y_2 \ldots Y_m \qquad (m \geqslant 1) \qquad (1)$$

ou

$$X \longrightarrow a \qquad (2)$$

avec $X, Y_i \in V_N$, $a \in V_T$. Il est commode d'exprimer les règles ci-dessus sous la forme :

$$(X,a) \longrightarrow Y_1\, Y_2 \ldots Y_m \qquad (1a)$$

ou

$$(X,a) \longrightarrow \lambda \qquad (2a)$$

λ étant la chaîne vide.

L'algorithme d'analyse utilise des piles et peut être brièvement décrit de la façon suivante :

a) Considérons le couple formé par la métavariable "X" située au sommet d'une pile et le prochain symbole de base dans la chaîne donnée lue de gauche à droite. Trois cas sont alors possibles :

1. Il n'existe dans la grammaire considérée qu'une règle du type :

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

Dans ce cas, on efface X et on empile la chaîne

$$Y_m\ Y_{m-1} \cdots Y_2\ Y_1 \quad ,$$

Y_1 étant au sommet. On répète alors le processus décrit en a). Lorsque la règle (X,a) est du type (2a), l'opération effectuée est équivalente à une élimination de X du sommet de la pile.

2. Il existe plusieurs règles (X,a). On crée alors une nouvelle pile pour chaque règle, en recopiant dans chacune le contenu de la pile originelle, et on répète pour chacune de ces piles le procédé décrit en a).

3. Il n'existe pas de règle (X,a) : on abandonne alors la pile en question.

b) Une analyse unique est achevée lorsqu'après une suite de traitements du type a), on a examiné tous les éléments de la chaîne donnée, et qu'une pile est vide. Si l'on désire trouver toutes les analyses possibles, la condition supplémentaire est qu'il ne reste plus de pile à considérer.

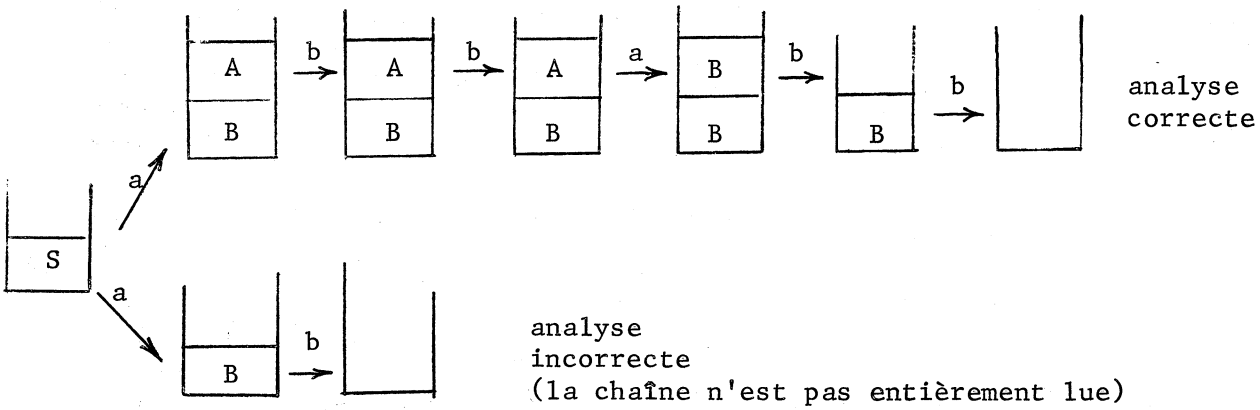
L'algorithme décrit ci-dessus commence avec une seule pile qui contient l'axiome S de la grammaire.

Exemple : Soit la grammaire écrite sous forme standard

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

$S \longrightarrow a\ A\ B$		$(S,a) \longrightarrow A\ B$
$S \longrightarrow a\ B$		$(S,a) \longrightarrow B$
$A \longrightarrow b\ A$	ces règles	$(A,b) \longrightarrow A$
$A \longrightarrow a\ B$	s'écrivent :	$(A,a) \longrightarrow B$
$B \longrightarrow a\ B$		$(B,a) \longrightarrow B$
$B \longrightarrow b$		$(B,b) \longrightarrow \lambda$

Soit à analyser la chaîne abbabb. La figure ci-dessous indique les contenus successifs des deux piles.



Signalons que la description que nous venons de présenter est théorique, et que deux piles suffisent pour la réalisation de cet algorithme en machine (le programme Algol correspondant est présenté en [Ndx]). Son utilisation dans le cadre d'un compilateur dirigé par la syntaxe est impensable actuellement pour les langages du type Algol. Ceci est dû au grand nombre de règles que l'on obtient lorsqu'on transforme la grammaire d'Algol en une autre dont les règles sont sous forme standard. NGUYEN-Dinh-Xuan a réalisé un programme (fondé sur l'algorithme de GREIBACH) effectuant automatiquement cette transformation dans le cas où la grammaire initiale ne contient pas de règles récursives à gauche [Ndx]. Appliqué à Algol, son programme produit des milliers de règles dont l'encombrement en mémoire est supérieur à la capacité des machines actuelles, à moins que l'on ne range ces règles sous forme de listes. Cependant l'algorithme de KUNO-OETTINGER

reste intéressant pour les langages qui se rapprochent des langues naturelles. L'équipe de Harvard a réussi à améliorer sensiblement les temps d'analyse en ajoutant à l'algorithme des vérifications supplémentaires qui permettent de réduire considérablement le nombre de retours en arrière ; ceci revient à augmenter la sélectivité de l'algorithme. L'un de ces moyens est le "shaper" que nous utilisons dans l'exemple du § V.5.1, et qui consiste à abandonner une pile lorsqu'elle contient plus de métavariables que la partie de la chaîne donnée qui reste à examiner. Nous verrons au paragraphe IV.3 une méthode de mise en ordre des règles syntaxiques particulièrement approprié à cet algorithme.

IV.2.3. Analyse ascendante non sélective

Nous présentons ici sommairement un algorithme initialement suggéré par COCKE et actuellement utilisé par le Centre d'Etudes pour la Traduction Automatique de Grenoble pour analyser les langues naturelles [VV]. Il se prête bien aux analyses multiples nécessaires à la détection des ambiguités dans les langues naturelles. Nous le présentons ici pour établir un parallèle entre les besoins d'analyse de la traduction automatique et ceux de la compilation. Son emploi effectif dans un compilateur de langage du type d'Algol serait absurde. Il pourrait cependant servir à l'analyse des langages spéciaux qui s'apparentent aux langues naturelles (cf. [Bob 2]).

Les règles de la grammaire doivent toutes être de l'une des formes suivantes :

$\Sigma_i \rightarrow \Sigma_j \Sigma_k$ règles de construction

$\Sigma_1 \rightarrow \rho_j$ règles lexicographiques

Σ et ρ, affectés d'un indice, représenteront respectivement un symbole non terminal et un symbole terminal.

On sait qu'à toute grammaire du type "context-free" G, on peut faire correspondre une grammaire G' ayant les propriétés indiquées et générant le même langage [Va, BC 2].

Exemple : Soit le vocabulaire terminal $V_T = \{a,b\}$, le vocabulaire non terminal $V_N = \{S,A,B\}$ et les règles suivantes constituant la grammaire G_0 :

S ⟶ aS
S ⟶ ABb
A ⟶ Aa
A ⟶ a
B ⟶ b

A cette grammaire correspond la grammaire suivante G_0' qui amène l'introduction des symboles C,D,E.

S ⟶ CS
S ⟶ DE
D ⟶ AB
A ⟶ AC
A ⟶ a
B ⟶ b
C ⟶ a
E ⟶ b

Pour faire fonctionner l'algorithme, on réécrit la grammaire obtenue sous forme d'une grammaire de reconnaissance G"(c'est-à-dire que $A \rightarrow \varphi$ est réécrite $\varphi \Rightarrow A$), et seules les règles de longueur deux sont rangées en machine dans un tableau M. La grammaire G pouvant contenir plusieurs règles ayant des sujets différents pour une même partie gauche, les éléments du tableau sont, en fait, des groupes de symboles ; pour permettre de repérer un des éléments du groupe, M aura une troisième dimension KM.

M [U, V, KM] sera égal au sujet de la KM-ième règle ayant comme partie droite l'élément U suivi de l'élément V.

Supposons que A, B et C aient les codes respectifs 1, 2 et 3 et que la grammaire G" comprenne les règles : $BC \Rightarrow B$ et $BC \Rightarrow A$,

alors M [2, 3, 1] = 2 (c'est-à-dire B)
M [2, 3, 2] = 1 (c'est-à-dire A).

Les règles de la grammaire étant de longueur 2, la structure de la chaîne sera représentée par un arbre binaire, c'est-à-dire, un arbre tel qu'à chaque noeud aboutissent deux branches. Les règles de longueur 1 servent seulement à transformer la chaîne donnée. A chacun de ses éléments ρ , on fait correspondre un ou plusieurs éléments non terminaux suivant qu'il existe, dans G", une ou plusieurs règles ayant ρ comme partie gauche. Par exemple, si G" contient les règles $\rho \Rightarrow \Sigma_i$ et $\rho \Rightarrow \Sigma_j$, à l'élément ρ de la chaîne d'entrée correspondront deux symboles Σ_i et Σ_j.

Comme zone de travail, l'algorithme utilise un tableau à trois dimensions T [I, J, K] dont chaque élément représente un noeud de l'arbre binaire que l'on cherche à obtenir. Un terme tel que T [I, J, K] représente le noeud auquel aboutit la K-ième structure associée à une sous-chaîne commençant au I-ème groupe de symboles et ayant la longueur J.

Nous appellerons "niveau" la deuxième dimension J de ce tableau : tous les groupes de symboles construits à partir des sous-chaînes de même longueur se trouvent sur une même ligne du tableau T. Nous pouvons en effet considérer le tableau T comme une matrice dont I repère la colonne et J la ligne. Chaque élément de cette matrice est un groupe de symboles, l'indice K repérant le K-ième d'entre eux.

L'algorithme utilise comme données le tableau M et la première ligne du tableau T, qui correspond à la chaîne donnée transformée, soit T [I, 1, K], I variant de 1 à L (longueur de la chaîne donnée).

Par niveaux successifs, l'algorithme calcule chacun des ensembles que constituent les éléments de la matrice T, de la manière suivante : les divers T [I, J, K] sont les sujets (s'ils existent) des règles ayant comme partie gauche :

soit T [I, 1, K1] suivi par T [I + 1, J - 1, K2] ,
soit T [I, 2, K1] suivi par T [I + 2, J - 2, K2] ,
. . .
. . .
. . .
. . .

soit $T\left[I, L1, K1\right]$ suivi par $T\left[I + L1, J - L1, K2\right]$,

soit $T\left[I, J-1, K1\right]$ suivi par $T\left[I + J - 1, 1, K2\right]$,

en donnant à K1 et K2 toutes les valeurs possibles.

Notons que la sous-chaîne de longueur J est bien composée de deux sous-chaînes dont la somme des longueurs est égale à J, et que l'indice K se nomme K1 ou K2 quand il appartient à un élément situé dans la partie gauche d'une règle.

La Figure 1 donne une représentation graphique du tableau T et de la façon dont on peut obtenir le groupe de symboles situés sur la ligne 5 et la colonne 2.

Le but de l'algorithme est la recherche des $T\left[1, L, K\right]$, c'est-à-dire des diverses structures associées à la chaîne de longueur L dont le premier élément est I = 1. Si K = 0, la chaîne donnée n'appartient pas au langage. Si K = 1, la chaîne possède une seule structure. Si $K > 1$, elle est ambiguë. Cette ambiguité peut provenir de l'application des règles de construction, dans l'algorithme, ou simplement des règles lexicographiques.

Nous présentons, en fin de paragraphe, le programme correspondant à cet algorithme écrit en Algol tel qu'il a été essayé sur calculateur. Pratiquement, dans le programme, l'épuisement des possibilités offertes par un ensemble $T\left[I, J, K\right]$ donné est repéré par l'existence

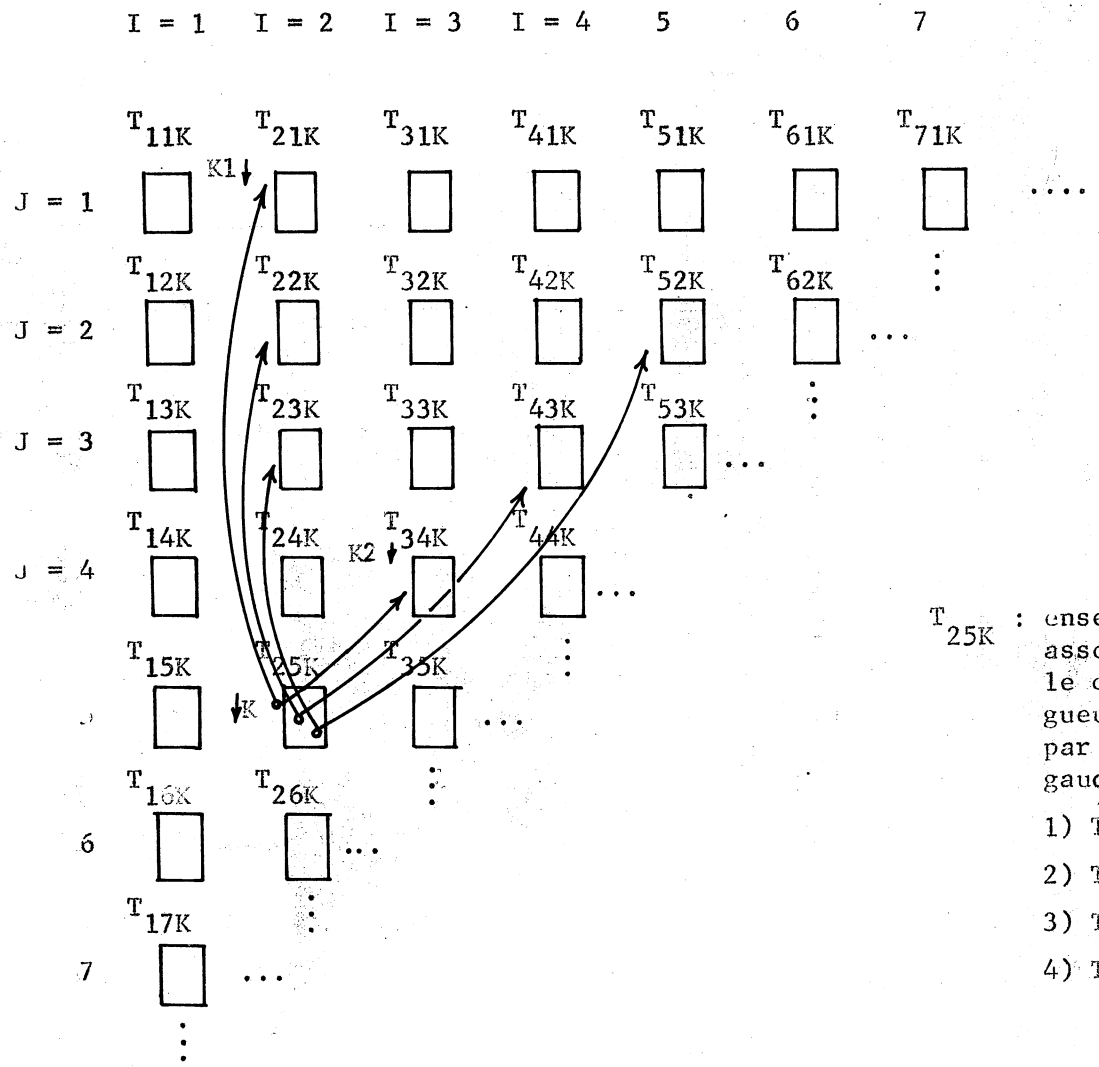


Figure 1

T_{25K} : ensemble des noeuds qui achèvent les structures associées aux sous-chaînes commençant par le deuxième élément (I = 2) et ayant la longueur 5 (J = 5). Cet ensemble est constitué par les sujets des règles ayant comme partie gauche

1) T_{21K1} suivi par T_{34K2}
2) T_{22K1} suivi par T_{43K2}
3) T_{23K1} suivi par T_{52K2}
4) T_{24K1} suivi par T_{61K2}

d'une marque notée ETOILE à l'endroit correspondant. Ceci s'applique également au tableau M. Nous avons déjà donné la signification des tableaux (entiers) M et T et des entiers KM, I, J, K1, K2, L1, L, ETOILE. Une mémoire ALPHA permet de ranger des résultats intermédiaires.

Nous n'avons pas cherché en écrivant ce programme à minimiser le nombre de mémoires utilisées ni le temps d'exécution. Le tableau M, qui est très creux, pourrait être rangée sous forme de liste, de même que la matrice T. L'ordre dans lequel sont calculés les ensembles T [I, J, K] peut être modifié (cf. [VV]) de manière à éviter certaines recherches inutiles et accroître ainsi la sélectivité de cet algorithme.

Nous présentons dans la Figure 2, écrite sous forme normale de Backus, la grammaire G_1 permettant de former des expressions arithmétiques. L'opérateur additif, l'opérateur multiplicatif, la variable et le nombre sont considérés comme des éléments du vocabulaire terminal(*). La table d'équivalence est donnée dans la Figure 2. La Figure 3 donne la transformation de G_1 en G_1', cette dernière ayant la forme désirée (cf. [BC 2]). Le tableau M est présentée en Figure 4. La chaîne donnée

$$\# \quad A = B * (C + 3) \quad \#$$

devient la suite :

#	v	e	v	m	g	v	p	n	d	#
B	C	D	C	I	K	C	H	E	J	B
	E		E			E		F		
	F		F			F		T		
	T		T			T				

(*) Cela peut être obtenu dans une phase d'édition préalable.

Le résultat final de l'analyse, constitué d'une seule structure, est présenté en Figure 5 : nous n'indiquons pas les intermédiaires qui n'aboutissent pas à une structure pour la chaîne complète.

Si l'on veut reconstituer l'arbre binaire directement par l'algorithme, il faut ranger en machine les éléments permettant de retrouver les deux composantes de la règle dont T [I, J, K] est sujet. On peut, par exemple, utiliser un tableau TB à quatre dimensions dans lequel sont notés L1, K1, K2 correspondant à I, J, K donnés :

TB [I, J, K, 1] := L1
TB [I, J, K, 2] := K1
TB [I, J, K, 3] := K2

Ces données suffisent à retrouver T [I, L1, K1] et T [I + L1, J - L1, K2], qui représentent les deux noeuds qui aboutissent au noeud T [I, J, K].

ALGORITHME D'ANALYSE MULTIPLE

```
pour J := 2 pas 1 jusqua L faire
    pour I := 1 pas 1 jusqua L - J + 1 faire
        début K := 1 ;
            pour L1 := 1 pas 1 jusqua J - 1 faire
                début K1 := 0 ;
                    pour K1 := K1 + 1 tantque T [I, L1, K1] ≠ ETOILE faire
                        début K2 := 0 ;
                            pour K2 := K2+1 tantque T [I+L1, J-L1,K2] ≠ ETOILE faire
                                début KM := 1 ;
                                    CODA : ALPHA := M[T[I,L1,K1], T[I+L1, J-L1,K2], KM] ;
                                        si ALPHA = ETOILE alors allera EPUISEE ;
                                        T[I, J, K] := ALPHA ;
                                        K := K + 1 ; KM := KM + 1 ;
                                        allera CODA ;
                            EPUISEE : fin
                            fin
                fin
        fin
```

Grammaire G_1

<programme> ::= # <suite d'instructions> #

<suite d'instructions> ::= <instruction> | <suite d'instructions> ; <instruction>

<instruction> ::= <variable> = <expression arithmétique simple>

<expression arithmétique simple> ::= <terme> | <expression arithmétique simple> <opérateur additif> <terme>

<terme> ::= <facteur> | <terme> <opérateur multiplicatif> <facteur>

<facteur> ::= <variable> | <nombre> | (expression arithmétique simple)

<opérateur additif> ::= + | -

<opérateur multiplicatif> ::= × | /

<variable> est une suite de lettres

<nombre> est une suite de chiffres

Table d'équivalences

<programme>	S
<suite d'instructions>	P
<instruction>	A
<expression arithmétique simple>	E
<terme>	T
<facteur>	F
;	s
<variable>	v
=	e
<opérateur additif>	p
<opérateur multiplicatif>	m
<nombre>	n
(	g
)	d

Figure 2

G_1	G_1^1	G_1^2	G_1'
$S \rightarrow \# P \#$	$S \rightarrow \# P \#$	$\begin{cases} S \rightarrow S_1 B \\ S_1 \rightarrow \# P \\ B \rightarrow \# \end{cases}$	$\begin{cases} S \rightarrow S_1 B \\ S_1 \rightarrow BP \end{cases}$ $B \rightarrow \#$
$P \rightarrow P s A$	$P \rightarrow P s A$		
$P \rightarrow A$	$P \rightarrow v e E$	$\begin{cases} P \rightarrow S_3 A \\ S_3 \rightarrow P s \end{cases}$	$\begin{cases} P \rightarrow S_3 A \\ S_3 \rightarrow PG \end{cases}$ $G \rightarrow s$
$A \rightarrow v e E$	$A \rightarrow v e E$		$\begin{cases} P \rightarrow S_2 E \\ S_2 \rightarrow C D \end{cases}$ $C \rightarrow v$ $D \rightarrow e$
$E \rightarrow E p T$	$E \rightarrow E p T$	$\begin{cases} P \rightarrow S_2 E \\ S_2 \rightarrow ve \end{cases}$	
$E \rightarrow T$	$E \rightarrow T m F$	$A \rightarrow S_2 E$	$A \rightarrow S_2 E$
$T \rightarrow T m F$	$E \rightarrow v$	$\begin{cases} E \rightarrow S_4 T \\ S_4 \rightarrow E p \end{cases}$	$\begin{cases} E \rightarrow S_4 T \\ S_4 \rightarrow E H \end{cases}$ $H \rightarrow p$
$T \rightarrow F$	$E \rightarrow n$		
$F \rightarrow v$	$E \rightarrow g E d$	$\begin{cases} E \rightarrow S_5 F \\ S_5 \rightarrow T m \end{cases}$	$\begin{cases} E \rightarrow S_5 F \\ S_5 \rightarrow T I \end{cases}$ $I \rightarrow m$
$F \rightarrow n$	$T \rightarrow T m F$	$E \rightarrow v$	$E \rightarrow v$
$F \rightarrow g E d$	$T \rightarrow v$	$E \rightarrow n$	$E \rightarrow n$
	$T \rightarrow n$	$\begin{cases} E \rightarrow S_6 J \\ S_6 \rightarrow g E \\ J \rightarrow d \end{cases}$	$\begin{cases} E \rightarrow S_6 J \\ S_6 \rightarrow KE \end{cases}$ $K \rightarrow g$
	$T \rightarrow g E d$		$J \rightarrow d$
	$F \rightarrow v$	$T \rightarrow S_5 F$	$T \rightarrow S_5 F$
	$F \rightarrow n$	$T \rightarrow v$	$T \rightarrow v$
	$F \rightarrow g E d$	$T \rightarrow n$	$T \rightarrow n$
		$T \rightarrow S_6 J$	$T \rightarrow S_6 J$
		$F \rightarrow v$	$F \rightarrow v$
		$F \rightarrow n$	$F \rightarrow n$
		$F \rightarrow S_6 J$	$F \rightarrow S_6 J$

Figure 3

		1 S	2 P	3 A	4 E	5 T	6 F	7 B	8 C	9 D	10 G	11 H	12 I	13 J	14 K	15 S_1	16 S_2	17 S_3	18 S_4	19 S_5	20 S_6
1	S																				
2	P										17										
3	A																				
4	E											18									
5	T												19								
6	F																				
7	B		15																		
8	C									16											
9	D																				
10	G																				
11	H																				
12	I																				
13	J																				
14	K				20																
15	S_1							1													
16	S_2				2 3																
17	S_3			2																	
18	S_4					4															
19	S_5						4 5														
20	S_6													4 5 6							

Tableau M

Figure 4

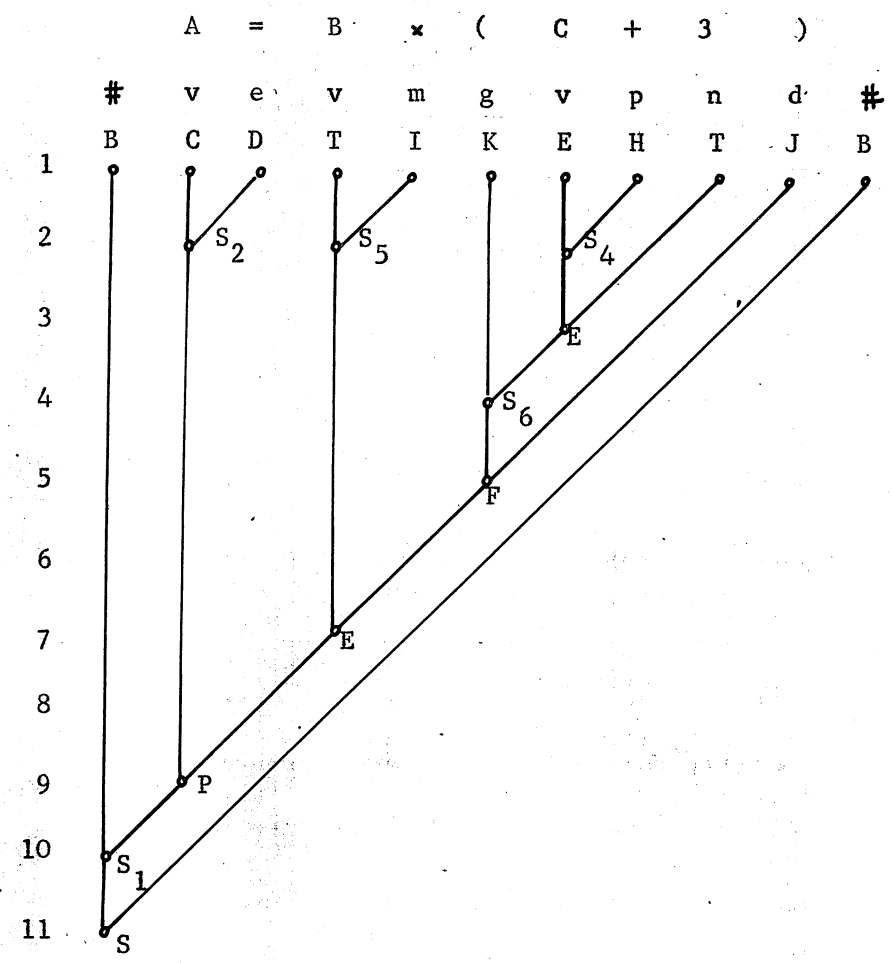

A = B × (C + 3)
# v e v m g v p n d #
B C D T I K E H T J B
1
2
3
4
5
6
7
8
9
10
11
S2
S5
S4
E
S6
F
E
P
S1
S

Figure 5

IV.2.4. Analyse ascendante sélective

L'algorithme présenté ici est assez proche de celui d'IRONS [Ir 1, May], écrit en Algol comme une procédure récursive dont l'exécution par un calculateur implique la gestion d'une pile par le compilateur existant sur cette machine.

La principale caractéristique de l'analyseur d'IRONS est le fait qu'il essaie, quand il rencontre des règles auto-imbriquées, de grouper le plus grand nombre possible d'éléments de la chaîne donnée sous le même symbole non terminal. Il peut parfois reculer quand la direction prise est mauvaise. Soit, par exemple, la grammaire comprenant les règles :

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

On convient de dire qu'il y a deux solutions commençant par b et conduisant au sujet A. Si l'algorithme ayant A comme but immédiat ne trouve pas a après b, il examinera la deuxième solution.

Dans d'autres cas, l'analyseur ne peut pas reculer. Considérons par exemple une chaîne telle que baaa construite à partir de la grammaire ci-dessous :

1) $S \longrightarrow Aa$
2) $A \longrightarrow Aa$
3) $A \longrightarrow b$

L'analyse passera par les étapes successives :

φ_2 = A a a a	en appliquant la règle 3)
φ_3 = A a a	en appliquant la règle 2)
φ_4 = A a	en appliquant la règle 2)
φ_5 = A	en appliquant la règle 2) alors qu'il faudrait appliquer la règle 1) pour obtenir S.

La chaîne baaa n'est donc pas reconnue alors qu'elle appartient au langage généré par la grammaire considérée. Elle le sera si l'on peut refuser la dérivation φ_5 et la remplacer par S. Remarquons que cette dernière limitation de l'algorithme d'IRONS ne semble pas concerner le langage Algol.

L'algorithme sélectif d'analyse ascendante présenté en Algol à la fin de ce chapitre est en revanche capable de traiter des cas analogues à celui que nous venons de considérer. La version présentée ici a été proposée par UNGER [Un] ; elle peut être facilement modifiée pour fournir des analyses multiples. Signalons également que l'idée émise par BASTIAN peut elle aussi être facilement ajoutée à cet algorithme [Bas].

Nous exprimerons encore la grammaire sous forme de reconnaissance. Pour les besoins de l'algorithme, nous transformons cette grammaire en un ensemble d'arbres, comme le montre l'exemple suivant :

Soit la grammaire G_2^* :

(*) Le langage décrit par cette grammaire est composé des chaînes $a^{m+n}ba^nb$ avec $m \geqslant 1$ et $n > 0$ (a^n représente la chaîne formée de a concaténé n-1 fois à lui-même : a^3 représente aaa).

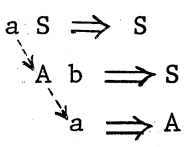
a S ⇒ S

a A b ⇒ S

b ⇒ A

a A a ⇒ A

Elle s'écrira en 2 arbres :



1er arbre à 2 branches, la deuxième ayant 2 sous-branches

b ⇒ A — 2ème arbre

La flèche pointillée relie un élément à celui qui le précède dans la règle. Dans le calculateur, les données correspondantes sont rangées dans trois tableaux à une dimension*, ARBRE, PRECEDANT, ALTERNANT d'indice J.

J	PRECEDANT	ARBRE	ALTERNANT
1	* *	a	*
2	1	S	5
3	2	S	*
4	*	*	*
5	1	A	*
6	5	b	9
7	6	S	*
8	*	*	*
9	5	a	*
10	9	A	*
11	*	*	*
12	* *	b	*
13	12	a	*
14	*	*	*

(*) Ces tableaux, ainsi que les piles explicites utilisées par l'algorithme doivent être déclarés comme des tableaux entiers.

ARBRE contient les éléments de la grammaire, préalablement codés, notés dans l'ordre de lecture des règles. Afin de repérer les fins de règles, nous les faisons suivre d'une marque "*", représenté par ETOILE dans le programme Algol.

PRECEDANT [J] indique quel élément précède, dans une règle donnée, l'élément J considéré ; pour l'élément le plus à gauche du symbole $\Rightarrow$, PRECEDANT [J] sera égal au symbole "**", représenté par DEUX ETOILES.

Considérons la représentation de la grammaire comme un ensemble d'arbres dont les racines sont les sujets des règles. Du tronc commun peuvent partir plusieurs branches. A la naissance de chacune d'elles, ALTERNANT [J] indique où se trouve, dans ARBRE, le début de la branche suivante. Il joue donc le rôle de la flèche pointillée. Dans ces deux derniers tableaux, comme dans les suivants, la marque ETOILE remplace les éléments sans signification spéciale.

Les données précédentes relatives à la grammaire sont insuffisantes pour les besoins de l'algorithme, qui doit distinguer les éléments du vocabulaire terminal des autres. Le tableau booléen TERMINAL répond à cette question. D'autre part, il est pratique de connaître l'endroit où commence dans ARBRE une règle donnée : le tableau RACINE permet d'obtenir la valeur correspondante de l'indice J.

Pour la grammaire G_2 donnée ci-dessus les tableaux TERMINAL et RACINE ont les valeurs suivantes* :

SYMBOLE	TERMINAL	RACINE
a	vrai	1
b	vrai	12
A	faux	*
S	faux	*

La sélectivité est ici introduite au moyen du tableau SUCCESSEUR qui indique si l'application d'une règle commençant par le symbole actuellement considéré peut conduire au but cherché. Le tableau SUCCESSEUR est une matrice booléenne carrée, de dimension égale au nombre de symboles du vocabulaire. SUCCESSEUR $[P,Q]$ a la valeur <u>vrai</u> si l'on peut trouver une suite $\alpha_1, \alpha_2, \ldots, \alpha_i, \ldots, \alpha_n$ telle que :

1) $\alpha_1 = P$; $\alpha_n = Q$; $\alpha_1 \in V_N \cup V_T$; pour tout $1 < i \leqslant n$, $\alpha_i \in V_N$

2) pour tout $1 \leqslant i < n$, il existe une règle $\alpha_i \omega \Rightarrow \alpha_{i+1}, \omega$ étant une chaîne sur $V_T \cup V_N$, peut-être vide.

De cette définition, il résulte la propriété transitive suivante :

si SUCCESSEUR $[P_1,P_2]$ et SUCCESSEUR $[P_2,P_3]$ sont tous deux <u>vrai</u>, alors SUCCESSEUR $[P_1,P_3]$ est <u>vrai</u>.

Nous donnons ci-dessous la matrice SUCCESSEUR correspondant à la grammaire G_2.

	a	b	A	S
a	faux	faux	vrai	vrai
b	faux	faux	vrai	faux
A	faux	faux	faux	faux
S	faux	faux	faux	faux

(*) Remarquons que dans le cas général un élément du vocabulaire non terminal peut aussi être racine d'un arbre.

Les tableaux mentionnés ici, à l'exception du dernier, peuvent s'obtenir facilement à partir de la description de la grammaire donnée sous FNB (nous avons effectivement programmé l'algorithme correspondant).

La détermination de la matrice SUCCESSEUR est plus complexe. Elle se fait à partir d'une matrice M contenant les suites α_1, α_2, obtenues directement à partir des règles de la grammaire.

SUCCESSEUR = $M_n = \bigcup_{k=1}^{n} M^K$ avec $M^1 = M$ et $M^{K+1} = M^K \cap M$ (*)

M^k donne les suites $\alpha_1, \ldots, \alpha_K$.

WARSHALL a montré qu'il existait une manière plus simple et équivalente de former SUCCESSEUR (cf. [Wa 2]). Dans la définition ci-dessus, la valeur de n est à déterminer.

Considérons la suite $M_1, M_2 = M^1 \cup M^2, \ldots, M_K = M^1 \cup M^2 \cup \ldots \cup M^K, M_{K+1}, \ldots, M_n$. Pour une certaine valeur de K, on a : $M_K \equiv M_{K+1}$. On prendra $M_n = M_K$.

WARSHALL a défini l'algorithme suivant, qui permet de déterminer M_n à partir de M_1 déjà appelé SUCCESSEUR. N est la dimension de la matrice.

(*) Par définition :

si $A \cap B = C$, C est tel que $C_{ij} = \bigcup_{1} a_{i1} \cap b_{1j}$

si $A \cup B = D$, D est tel que $d_{ij} = a_{ij} \cup b_{ij}$

$\bigcup_{k=1}^{n} M^K = M^1 \cup M^2 \cup \ldots \cup M^K \cup \ldots \cup M^n$.

```
pour J := 1 pas 1 jusqua N faire
   pour I := 1 pas 1 jusqua N faire
     début
       si SUCCESSEUR [I,J] alors
          début
             pour L := 1 pas 1 jusqua N faire
               SUCCESSEUR [I,L] := SUCCESSEUR [I,L] ∪ SUCCESSEUR [J,L]
          fin
     fin ;
```

La matrice SUCCESSEUR étant en général très creuse, il est intéressant de la ranger sous forme de liste, ou d'utiliser chaque position binaire d'un mot machine pour indiquer le caractère vrai ou faux d'un élément de SUCCESSEUR. C'est ainsi que procède IRONS dans son compilateur [Ir 4].

Les six tableaux précédemment décrits représentent la grammaire du langage dont on veut analyser les chaînes terminales. Ces dernières sont rangées en machine sous forme d'un tableau à une dimension : CHAINE [I] a pour valeur le code numérique correspondant à l'élément du vocabulaire terminal occupant la I-ième position de la chaîne.

L'analyseur que nous décrivons ici utilise deux piles explicites. Nous avons choisi l'utilisation explicite des piles par opposition avec les algorithmes où la récursivité les utilise implicitement (cf. [Ir 1], § V.5.1) ; ceci permet de préciser le mécanisme de retour en arrière. L'une des piles, BUT, contient les buts cherchés, chacun suivi de deux indices dont la signification est donnée ci-dessous. L'autre, TROUVE,

renferme les sujets des règles utilisées, correspondant aux buts obtenus, ainsi que d'autres éléments, décrits plus loin, permettant de reconstituer la structure de la chaîne donnée.

Examinons sur un exemple le fonctionnement de l'algorithme.

Exemple : Cherchons si la chaîne aabb appartient au langage décrit par la grammaire G_2.

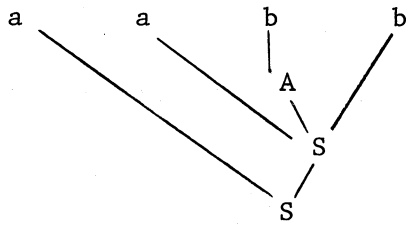
I	: 1	2	3	4
CHAINE [I]	: a	a	b	b

Au départ, I = 1. SUCCESSEUR [a,S] étant <u>vrai</u>, RACINE [a] = 1 donne l'adresse dans ARBRE des règles commençant par a. Si l'on cherche à appliquer la première d'entre elles, aS $\Rightarrow$ S, S, non sujet, doit suivre a. Notre objectif S est placé dans la pile BUT, tandis que le pointeur se déplace d'une unité vers la droite. Un déroulement identique au précédent conduit à empiler une nouvelle fois S dans BUT et à repérer b = CHAINE [3].

SUCCESSEUR [b,S] étant <u>faux</u>, le but S n'est pas à poursuivre.

ALTERNANT [2] étant égal à 5, nous chercherons à transformer b en A = ARBRE [5]. Ceci est possible car SUCCESSEUR [b,A] est <u>vrai</u>. Le nouvel objectif A est empilé dans BUT. RACINE [b] indique la règle b $\Rightarrow$ A, transforme b en A qui, de BUT, est transféré dans TROUVE. L'algorithme cherche à obtenir les sous-chaînes qu'il est possible de remplacer par A, dans le cas où SUCCESSEUR [A,A] est <u>vrai</u> ; ceci ne se produit pas avec la grammaire G_2. Le sommet de la pile BUT contient maintenant S. Rappelons que A était à former pour la règle aAb $\Rightarrow$ S

et que b = ARBRE [6] est à comparer à b = CHAINE [4], CHAINE [3] ayant servi à former A. Les deux symboles b dans ARBRE et dans CHAINE étant identiques, la règle aAb ⇒ S peut être appliquée. S est transféré de BUT dans TROUVE et l'on revient à la règle aS ⇒ S ; nous avons tous les éléments de la partie gauche de celle-ci. Nous pouvons appliquer la règle et transférer S de BUT dans TROUVE. La pile BUT est alors vide, tous les symboles de la chaîne d'entrée sont analysés et nous avons l'arbre suivant, qu'il est possible de reconstituer à partir de la pile TROUVE.



Dans l'algorithme sont également utilisées les deux procédures suivantes :

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

```
procédure EMPILER TROUVE (T1,T2,T3,T4,T5) ; entier T1,T2,T3,T4,T5 ;
    début entier V ;
      pour V := T1,T2,T3,T4,T5 faire
           début
               TROUVE [L] := V ;
               L := L + 1
           fin
    fin procédure EMPILER TROUVE ;
```

EMPILER BUT permet en une seule instruction de stocker dans la pile BUT le symbole non terminal que l'on cherche à former suivi par le pointeur I de la chaîne donnée et par le pointeur J du tableau ARBRE, correspondant à l'endroit où ce symbole vient d'être rencontré. EMPILER TROUVE permet de ranger dans la pile TROUVE cinq éléments successifs. Les trois premiers proviennent de la pile BUT ; ils sont suivis du pointeur I de CHAINE au moment où la fin de la règle est rencontrée et du pointeur J correspondant à cette fin de règle dans ARBRE.

Nous présentons successivement, à la fin de ce paragraphe, l'algorithme lui-même, puis les productions de la grammaire G_1 du § IV.2.3, écrite sous forme de reconnaissance puis comme un ensemble d'arbres, ainsi que les tableaux ARBRE, PRECEDANT et ALTERNANT correspondants. La Figure 7 donne les tableaux TERMINAL, RACINE et SUCCESSEUR formés à partir de G. La Figure 8 montre la chaîne donnée :

A = B x (C + 3) ; D = 4

réécrite ainsi :

v e v m g v p n d s v e n

Elle donne également la structure de cette chaîne construite à partir du contenu de la pile TROUVE présenté sur la partie gauche.

ALGORITHME D'ANALYSE ASCENDANTE

```
INITIALISATION : K := L := I := J := 1 ;
                 EMPILER BUT (AXIOME, I, MARQUE) ;
                 si ¬ SUCCESSEUR [CHAINE [I], AXIOME] alors allera ERREUR ;

RENTREE ARBRE :  J := RACINE [CHAINE [I]] ;

VERIFIER SI TERMINAL :
                 J := J + 1 ; I := I + 1 ;
                 si TERMINAL [ARBRE [J]] alors
                    allera si CHAINE [I] = ARBRE [J] alors VERIFIER SI TERMINAL
                                                     sinon RECULER ;
                 si ARBRE [J + 1] = ETOILE alors
                    début
                       I := I - 1 ;
                       si ARBRE [J] = BUT [K - 3] alors
                          début
                             EMPILER TROUVE (BUT [K - 3], BUT [K - 2], BUT [K - 1],
                                                                       I, J) ;
                             si BUT [K - 3] = AXIOME alors allera TERMINE ;
                 ESSAYER EXPANSION :
                              si SUCCESSEUR [BUT [K - 3], BUT [K - 3]] alors
                                 début
                                    J := RACINE [BUT [K - 3]] ;
                                    allera VERIFIER SI TERMINAL
                                 fin ;
                 ENLEVER BUT : J := BUT [K - 1] ;
                               K := K - 3 ;
                               allera VERIFIER SI TERMINAL
                          fin ;
                       si ¬ SUCCESSEUR [ARBRE [J], BUT [K-3]] alors allera RECULER2;
                       EMPILER TROUVE (ARBRE [J], BUT [K - 2], 0, I, J) ;
                       J := RACINE [ARBRE [J]] ;
                       allera VERIFIER SI TERMINAL
                    fin ;
                 si ¬ SUCCESSEUR [CHAINE [I], ARBRE [J]] alors allera RECULER ;
                 EMPILER BUT (ARBRE [J], I, J) ;
                 allera RENTREE ARBRE ;

RECULER :        I := I - 1 ;
```

```
RECULER2 :      si ALTERNANT [J] ≠ ETOILE alors
                   début
                       J := ALTERNANT [J] - 1 ;
                       allera VERIFIER SI TERMINAL
                   fin ;
                J := PRECEDANT [J] ;
                si ¬ TERMINAL [ARBRE [J]] ∧ PRECEDANT [J] ≠ DEUX ETOILES alors
                   début
                      EMPILER BUT (TROUVE [L - 5], TROUVE [L - 4], TROUVE [L - 3]) ;
                      ENLEVER TROUVE :
                      J := TROUVE [L - 1] ;
                      L := L - 5 ;
                      allera RECULER2
                   fin ;
                si ¬ TERMINAL [ARBRE [J]] ∧ PRECEDANT [J] = DEUX ETOILES alors
                   allera si BUT [K - 3] = TROUVE [L - 5] alors ENLEVER BUT
                                                          sinon ENLEVER TROUVE ;
                si PRECEDANT [J] ≠ DEUX ETOILES alors allera RECULER ;
                J := BUT [K - 1] ;
                K := K - 3 ;
                allera si J = MARQUE alors ERREUR sinon RECULER ;
```

Remarques : Les étiquettes TERMINE et ERREUR indiquent respectivement la fin d'un programme et un renvoi à un message d'erreur. K et L sont les indices des piles BUT et TROUVE.

J	PRECEDANT	ARBRE	ALTERNANT
1	* *	#	*
2	1	P	*
3	2	#	*
4	3	S	*
5	*	*	*
6	* *	P	*
7	6	s	*
8	7	A	*
9	8	P	*
10	*	*	*
11	* *	A	*
12	11	P	*
13	*	*	*
14	* *	v	*
15	14	e	34
16	15	E	*
17	16	A	*
18	*	*	*
19	* *	E	*
20	19	p	*
21	20	T	*
22	21	E	*
23	*	*	*
24	* *	T	*
25	24	E	27
26	*	*	*
27	24	m	*
28	27	F	*
29	28	T	*
30	*	*	*
31	* *	F	*
32	31	T	*
33	*	*	*
34	14	F	*
35	*	*	*
36	* *	n	*
37	36	F	*
38	*	*	*
39	* *	g	*
40	39	E	*
41	40	d	*
42	41	F	*
43	*	*	*

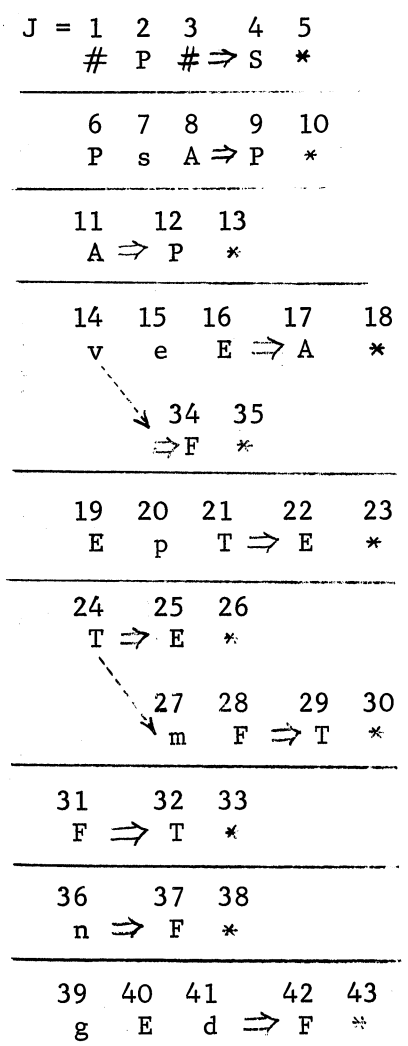
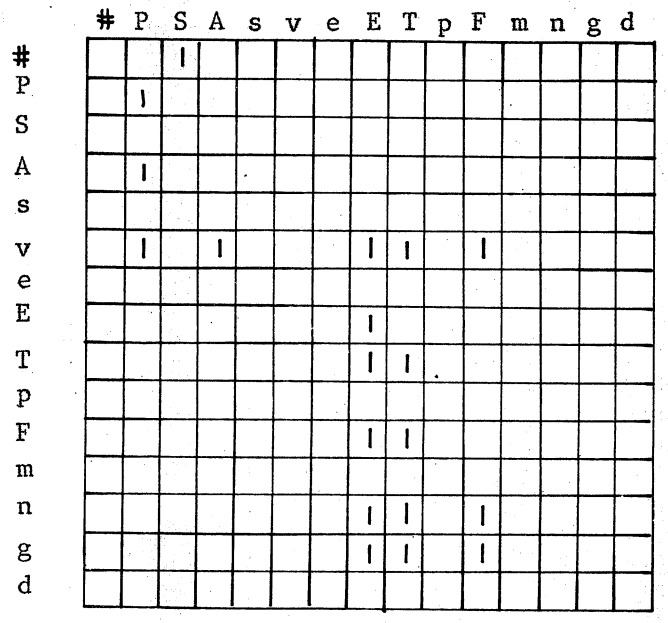


Figure 6

SYMBOLE	TERMINAL	RACINE
#	I	1
P	O	6
S	O	*
A	O	11
s	I	*
v	I	14
e	I	*
E	O	19
T	O	24
p	I	*
F	O	31
m	I	*
n	I	36
g	I	39
d	I	*

I signifie vrai
O signifie faux



I signifie vrai
blanc signifie faux

SUCCESSEUR

Figure 7

L	TROUVE L (SUJET)	TROUVE L+1 (I initial)	TROUVE L+3 (I final)	TROUVE L+4 (J)
1	F	4	4	34
6	T	4	4	32
11	F	7	7	34
16	T	7	7	32
21	E	7	7	25
26	F	9	9	37
31	T	9	9	32
36	E	7	9	22
41	F	6	10	42
46	T	4	10	29
51	E	4	10	25
56	A	2	10	17
61	P	2	10	12
66	F	14	14	37
71	T	14	14	32
76	E	14	14	25
81	A	12	14	17
86	P	2	14	9
91	S	1	15	4

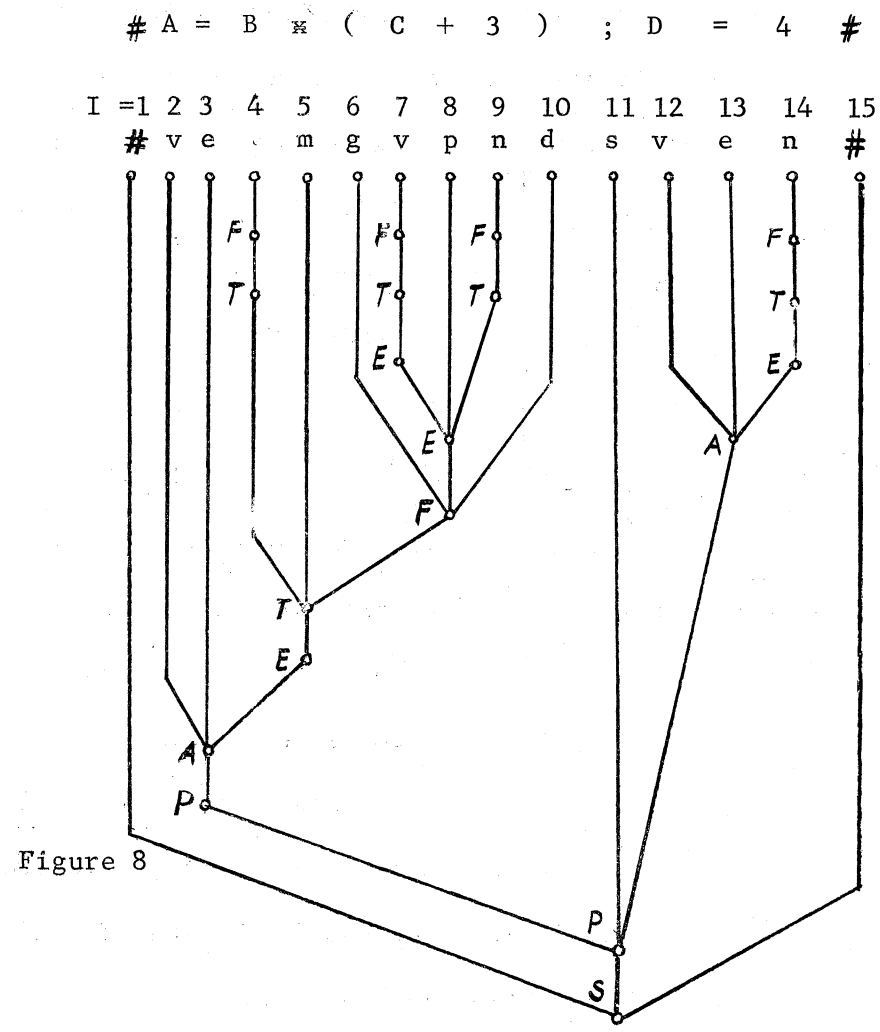


Figure 8

IV.3. Mise en ordre des règles syntaxiques

Nous avons vu que dans le cas des analyseurs non déterministes l'analyse se fait par essais et erreurs : un retour en arrière est inévitable quand on choisit une règle qui par la suite se montre inapplicable. La mise en ordre des règles à essayer peut donc affecter considérablement la vitesse d'analyse et par conséquent l'efficacité du compilateur. IRONS [Ir 4] a remarqué que la solution du problème de l'ordre optimal des règles n'était pas immédiat et suggère qu'il soit laissé "à la méditation pour les jours de pluie". A notre connaissance, il n'existait pas de solution mathématique à ce problème ; nous en proposons une à la fin de ce chapître.

La solution présentée ici est essentiellement pratique et consiste à enregistrer d'abord le nombre d'utilisations d'une règle couronnées de succès. Dans l'étape suivante et au choix de l'utilisateur le programme réordonne les règles pour que les premières considérées dans une analyse ultérieure soient les plus fréquemment utilisées. La mise en ordre des règles est particulièrement importante dans l'algorithme d'analyse de KUNO-OETTINGER (cf. § IV.2.2) lorsque l'on ne s'intéresse qu'à une seule analyse, puisque le nombre de règles d'une grammaire sous forme "standard" est en général très supérieur à celui de la grammaire "context-free" originelle qui lui est équivalente. De plus le processus de remise en ordre proposé ici est très facilement incorporable à cet algorithme ; on peut réaliser des schémas analogues pour d'autres méthodes d'analyse.

Organisation de la mémoire et modes de fonctionnement

Soit POINTEUR $[X,a]$ un tableau où l'on range les pointeurs $p^1_{X,a}$. Les indices "X" et "a" correspondent aux codes des éléments "X" et "a" d'une règle écrite sous la forme

$$(X,a) \longrightarrow Y_1 Y_2 Y_3 \ldots\ldots Y_m \qquad (1)$$

ou

$$(X,a) \longrightarrow \Lambda \qquad (2)$$

Ces pointeurs sont à leur tour les indices d'un autre tableau à une dimension TABLE où l'on range les parties droites des règles de la façon suivante :

TABLE $[p^i_{X,a}]$	contient	$p^{i+1}_{X,a}$
TABLE $[p^i_{X,a} + 1]$	contient	m^i
TABLE $[p^i_{X,a} + 2]$	contient	$\sum^i_{X,a}$
TABLE $[p^i_{X,a} + 3]$	contient	Y^i_1
TABLE $[p^i_{X,a} + 4]$	contient	Y^i_2
.		
.		
.		
TABLE $[p^i_{X,a} + m^i + 2]$	contient	$Y^i_{m^i}$

L'indice supérieur i indique la i-ème règle du type (1) ou (2) pour un couple (X,a) ; $\sum^i_{X,a}$ représente la fréquence d'utilisation de cette règle POINTEUR $[X,a]$ contient, lorsque la règle (X,a) correspondante n'existe pas, une marque spéciale dont la présence peut être vérifiée

par l'analyseur. De même lorsque $p^{i+1}_{X,a}$ est un symbole spécial, l'information qui le suit en TABLE représente la dernière des règles (X,a). Quand m^i=0 la règle est du type (2).

Décrivons maintenant les modes de fonctionnement possibles. Le mode "analyse d'échantillons" permet l'analyse des chaînes au moyen d'une configuration arbitraire des $p^i_{X,a}$. Après chaque analyse on évalue les $\sum^i_{X,a}$. Le mode "remise en ordre" permet de changer les $p^i_{X,a}$ de façon à ce que $p^1_{X,a}$ corresponde au plus grand des $\sum^i_{X,a}$; les $p^i_{X,a}$ sont rectifiés de façon à correspondre aux $\sum^i_{X,a}$ en ordre décroissant. L'utilisateur peut aussi choisir le mode "analyse normale" dans lequel les $p^i_{X,a}$ (et $\sum^i_{X,a}$) restent inchangés.

Le programme de mise en ordre que nous venons de décrire a été mis au point par NGUYEN-Dinh-Xuan [Ndx]. Afin de vérifier les gains de temps obtenus par la remise en ordre des règles nous avons considéré une partie de la syntaxe d'Algol et nous avons mesuré les temps d'analyse. Ces gains dépendent évidemment des analogies structurelles entre les chaînes échantillons et celles à analyser. Les analyses postérieures à la mise en ordre ont pris de 1/2 à 1/20 du temps d'une analyse normale (c'est-à-dire avec une configuration arbitraire des $p^i_{X,a}$). Nous avons aussi vérifié que certains $\sum^i_{X,a}$ avaient des valeurs identiques, un deuxième niveau d'optimisation étant possible en vue d'améliorer encore plus la vitesse de l'analyse.

Remarquons pour terminer que la méthode suggérée dans ce paragraphe, le système utilisé par GARWICK pour les tableaux à dimension variable (cf. [Gar 2] et § III.4.1) et le schéma d'utilisation des mémoires secondaires de l'ordinateur Atlas (cf. [Ki] et Appendice A) sont tous les trois issus du même principe : un apprentissage même rudimentaire peut réduire considérablement le temps d'exécution des programmes dans lesquels la variation de certaines quantités est a priori imprévisible mais se précise au fur et à mesure du déroulement du programme.

Sujet de recherche

On pourrait faire une étude théorique de la mise en ordre au moyen de matrices de probabilité au sens de Markov : l'élément p_{ij} donne la probabilité du passage d'un élément syntaxique (ou état) i à l'élément j pendant l'analyse. Par exemple cette matrice pourrait être facilement calculée (par programme bien sûr !) pour l'ensemble des algorithmes Algol publiés dans les Communications de l'A.C.M. Si l'on disposait d'une telle matrice pour les symboles de base d'Algol il serait facile d'ordonner les règles selon la succession la plus probable de deux symboles de base. Un problème plus intéressant serait d'obtenir plusieurs matrices, une pour chaque succession de métavariables et de procéder à la mise en ordre selon les suites les plus probables dans tous les niveaux. On pourrait modifier un analyseur de façon à calculer ces matrices, ce qui constituerait sans doute un

projet intéressant. Il nous semble qu'une matrice de Markov établie pour les symboles de base d'Algol, à partir des programmes les plus courants, aiderait à construire des compilateurs plus efficaces même dans le cas d'écriture "à la main".

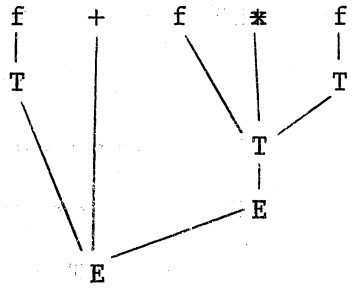
IV.4. Remarques sur l'analyse syntaxique

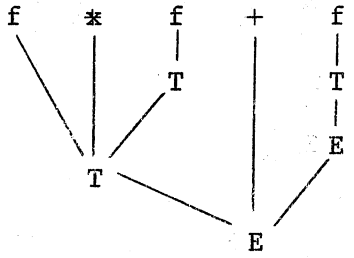
Ayant réalisé une analyse par l'une des méthodes que nous venons de décrire, nous devons aborder les questions suivantes : dans quelle mesure les métavariables sont-elles importantes pour la génération ? Comment utiliser le résultat de l'analyse pour générer un programme objet sous différentes formes ? La première question est liée au problème des transformations de grammaires : souvent celles-ci introduisent ou enlèvent certaines métavariables bien que la quantité d'information syntaxique semble être inchangée.

Considérons la grammaire G_1 :

$$E \longrightarrow T \mid T + E$$
$$T \longrightarrow f \mid f * T$$

où E représente une <expression>, T un <terme>, et f un <facteur>. Les deux analyses :

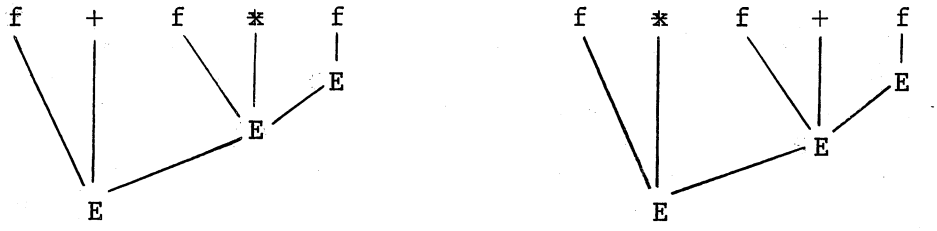




nous montrent que les noeuds relatifs au symbole + sont toujours au-dessous des noeuds relatifs au symbole *. En revanche la grammaire G_2

$$E \longrightarrow f \mid f * E \mid f + E$$

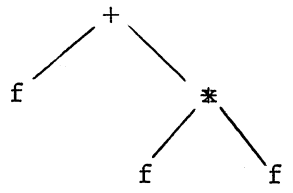
qui génère le même langage, fournit les analyses :



où cette propriété ne se vérifie pas. La position relative des méta-variables dans l'arbre d'analyse peut donc être appelée à jouer un rôle important dans le contenu sémantique d'une grammaire. Ainsi le fait qu'un noeud relatif au signe * soit toujours au-dessus d'un noeud voisin relatif au signe + sert à indiquer que l'opération * doit se réaliser avant l'opération +.

Le moyen que nous avons choisi pour relier l'analyse à la génération est la transformation de l'arbre d'analyse syntaxique en un autre où l'on regroupe les éléments de la chaîne donnée afin de mettre en évidence les opérateurs et leurs opérandes. En termes linguistiques, nous proposons la transformation de l'arbre chomskien en un arbre de dépendance (cf. [Gai, HSS]) ; en langage courant ceci revient à dire que nous proposons la transformation de l'analyse grammaticale en analyse logique. L'analogie entre les langues naturelles et les langages de programmation est d'ailleurs évidente : un "verbe" correspond à un opérateur ayant plusieurs opérandes (pronoms, substantifs, etc.) qui peuvent à leur tour contenir des opérateurs (propositions). Retournons à notre premier exemple : l'arbre

de dépendance correspondant à la chaîne donnée f + f * f est :

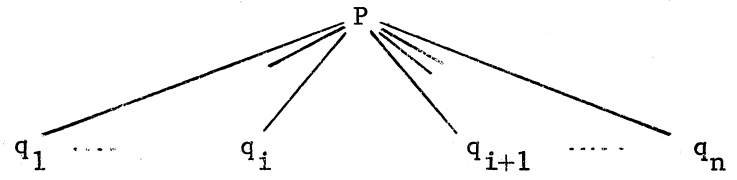


Observons que tous les noeuds de cet arbre appartiennent à V_T. Les grammaires de dépendance ont été initialement proposées par les linguistes HAYS et LECERF en 1960 [HSS]. Elles ont été étudiées du point de vue mathématique par GAIFMAN [Gai], qui a montré qu'elles ont la même puissance que les langages "context-free", si l'on y ajoute une restriction, dite de projectivité et décrite ci-dessous.

Les règles d'une grammaire de dépendance se présentent sous la forme

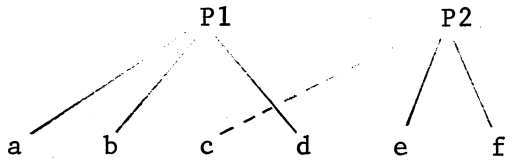
$$P\ (q_1, q_2, \ldots\ldots, q_i \wedge q_{i+1}, \ldots q_n) \quad ,$$

où P est un opérateur, les q_i sont des opérandes, et $\wedge$ est une marque qui indique que les éléments $q_{i+1}, \ldots q_n$ doivent obligatoirement être à droite de P (noeud principal), et que les q_1 à q_i doivent être à sa gauche. Autrement dit, une règle de dépendance représente un noeud et les branches qui en sortent :



La restriction de projectivité consiste à interdire les croisements

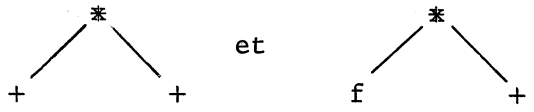
de branches dans une analyse. Par exemple, l'analyse ci-dessous est impossible :



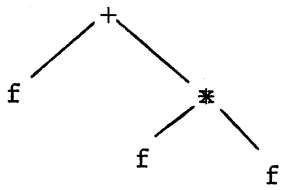
La grammaire G_1 peut facilement être mise sous forme de règles de dépendance, qui sont :

$$+ (f \quad f) \qquad + (+ \wedge f)$$
$$+ (f \quad *) \qquad + (+ \wedge *)$$
$$+ (* \quad f) \qquad * (* \wedge f)$$
$$+ (* \wedge *) \qquad * (f \wedge f)$$

Ceci interdit les configurations du type



La chaîne f + f * f fournit l'analyse :

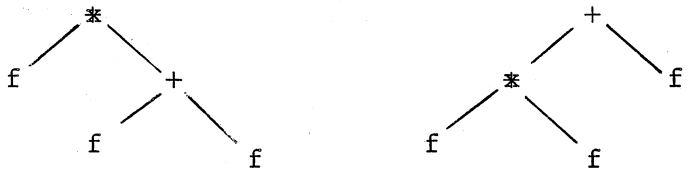


La grammaire G_2 est en revanche représentée par les règles ci-dessus et par

$$* (+ \wedge f)$$
$$* (f \wedge +)$$
$$* (+ \wedge +)$$
$$* (+ \wedge *)$$

$$* \; (* \wedge +)$$
$$* \; (f \wedge *)$$
$$* \; (* \wedge *)$$
$$+ \; (+ \wedge +)$$
$$+ \; (f \wedge +)$$
$$+ \; (* \wedge +)$$

Elle fournit pour la chaîne donnée f * f + f les deux analyses suivantes :



(a) (b)

Cela montre les deux sens que l'on peut attribuer à la chaîne donnée lorsqu'on n'affecte pas de priorité aux opérateurs * et + (cf. [Bol, Fl 4]).

La transformation d'arbres que nous proposons n'est qu'une simplification du formalisme suggéré par IRONS [Ir 1, Ir 3, Ir 4]. Cependant, cette simplification permet non seulement un allègement du langage de génération, mais aussi une indépendance du compilateur vis-à-vis de la méthode d'analyse employée.

Afin de réaliser la transformation d'un arbre d'analyse en un arbre de dépendance, nous attachons à chaque règle $A \longrightarrow \varphi$ de la grammaire C.F. une liste d'entiers positifs

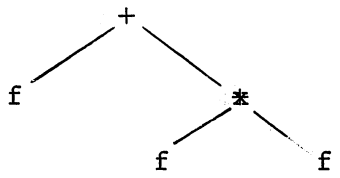
$$l_\varphi = (n_0, n_1, \ldots n_i, \ldots n_m),\ m < \text{longueur } (\varphi) \ ;$$

n_i correspond au n_i-ième élément de φ compté à partir de la gauche.

Le rang i ($i \geqslant 1$) dans 1_φ indique la i-ième nouvelle branche (correspondant au n_i-ième élément de φ), n_0 étant un noeud de l'arbre de dépendance. Si le n_i-ième élément de 1_φ correspond à un symbole de base, on le prend tel qu'il est ; s'il correspond à une métavariable B, on applique récursivement ce que nous venons de décrire à la règle $B \longrightarrow \psi$ et à sa liste d'entiers 1_ψ. Ainsi si l'on attache à G1 les listes suivantes :

$$E \longrightarrow T\ (1)$$
$$E \longrightarrow T + E\ (2\ 1\ 3)$$
$$T \longrightarrow f\ (1)$$
$$T \longrightarrow f * T\ (2\ 1\ 3)$$

on obtient pour la chaîne f + f * f (déjà analysée syntaxiquement plus haut) l'arbre de dépendance suivant :



Compte tenu du résultat final de notre analyse, on peut se demander s'il n'était pas possible d'utiliser directement les règles de dépendance et de procéder à une analyse fondée sur ces règles. Il nous semble que le nombre de règles de dépendance pour un langage tel qu'Algol est beaucoup plus important que celui des règles sous FNB ; cela n'est pourtant qu'une hypothèse qui reste à étudier plus à fond. Il ne nous semble pas non plus qu'il existe actuellement d'algorithme opérationnel et efficace d'analyse de dépendance.

Sujets de recherche

a. Ecriture de la grammaire d'Algol sous forme de dépendance. Compte tenu du nombre considérable de règles résultantes, il sera peut-être nécessaire d'introduire de nouveaux métasymboles pour exprimer, par exemple, l'union d'un ensemble d'opérandes. L'écriture d'un programme d'analyse fondé sur des règles de dépendance permettra de comparer son efficacité avec celle de l'analyse chomskienne.

b. Transformation de grammaires et de leurs listes numériques associées. Si l'on veut rendre le résultat final de notre analyse (c'est-à-dire l'arbre de dépendance) indépendant des règles d'une grammaire C.F. initiale, toutes les transformations réalisées sur ses règles doivent également modifier les listes d'entiers correspondantes. La transformation d'une grammaire sous forme normale ou sous forme standard sont des cas où une évaluation automatique de nouvelles listes serait souhaitable. Un problème lié à celui-ci mais dont la solution nous semble difficile est la détermination de la grammaire C.F. équivalente à une autre donnée et ayant le nombre minimum de règles qui permettent l'obtention du même arbre de dépendance.

IV.5. Langage d'analyse

IV.5.1. Syntaxe

⟨programme⟩ ::= ⟨règle⟩ | ⟨règle⟩ ⟨programme⟩
⟨règle⟩ ::= ⟨partie gauche⟩ ::= ⟨partie droite⟩ (⟨liste d'atomes⟩)
⟨partie gauche⟩ ::= ⟨métavariable⟩ | ⟨métavariable⟩ G
⟨partie droite⟩ ::= ⟨élément⟩ | ⟨élément⟩ ⟨blancs⟩ ⟨partie droite⟩
⟨élément⟩ ::= ⟨métavariable⟩ | ⟨symbole de base⟩ | ⟨suite 01.⟩
⟨liste d'atomes⟩ ::= ⟨atome⟩ | ⟨atome⟩ ⟨blancs⟩ ⟨liste d'atomes⟩
⟨blancs⟩ ::= ␣ | ␣ ⟨blancs⟩

Les ⟨métavariables⟩ sont des identificateurs au sens d'Algol. Les ⟨symboles de base⟩ sont des symboles de base du langage source. Une ⟨suite 01.⟩ a été définie au § III.2.1 et un ⟨atome⟩ le sera au § V.1. Syntaxiquement, l'atome utilisé ici est soit un entier sans signe, soit un identificateur au sens d'Algol, soit un symbole.

IV.5.2. Sémantique

La syntaxe définie ci-dessus n'est que celle de la FNB, à laquelle nous ajoutons les listes numériques permettant la transformation de l'arbre d'analyse en un arbre de dépendance, de la façon décrite au § IV.4. La distinction entre les ⟨métavariables⟩ et les ⟨symboles de base⟩ se fait par programme. Une généralisation a été apportée à la liste numérique décrite au § IV.4 : elle consiste à accepter des éléments non

numériques ; ceux-ci sont recopiés à la place correspondante lors de la transformation de l'arbre syntaxique en arbre de dépendance. Le symbole G définit un saut vers le programme de génération (cf. § VI.2). Ce saut s'effectue automatiquement lorsque l'analyse de la métavariable qui précède ce symbole est complète ; on construit d'abord l'arbre de dépendance correspondant avant de donner le contrôle au programme de génération.

IV.5.3. Exemple

```
identificateur ::= b 001.1 .......
facteur ::= identificateur  (1)
facteur ::= (expression)  (2)
terme ::= facteur * terme  (*  1  3)
terme ::= facteur  (1)
expression ::= terme  (1)
expression ::= terme + expression  (2  1  3)
axiome G ::= expression  (1)
```

Un autre exemple plus détaillé sera donné au § V.5.4.

V. Langage pour la manipulation des arbres

"Dans un paysage désolé près d'un
fleuve qui coule entre les ruines,
l'antique Dragon, maître des eaux
du devenir, se mord la queue.
Tout se transforme mais rien ne meurt,
enseigne la philosophie hermétique.
Chaque fin constitue un nouveau
commencement."

(R. Alleau, Histoire des Sciences
Occultes)

Nous proposons dans ce chapitre une inclusion de Lisp* dans Algol, c'est-à-dire que nous présentons un jeu de procédures qui permettent la simulation de Lisp en Algol. Nous appellerons par la suite Lisp-Algol les programmes en Algol qui utilisent ce jeu de procédures. Ils ont été utilisés à Grenoble depuis deux ans pour la solution de plusieurs problèmes de manipulation de symboles : détermination des coefficients des équations de la méthode de Runge-Kutta, calcul analytique d'erreurs dans les formules algébriques et trigonométriques, etc. L'adjonction

---

(*) Ce langage est décrit en [Mc 2, BB].

de Lisp à Algol s'est avérée indispensable pour de nombreuses applications dans le domaine des langages et de la compilation : nous en présenterons quelques-unes à la fin de ce chapitre.

Signalons que les procédures présentées ici ne constituent pas la seule façon d'incorporer un langage de listes à Algol. Il reste cependant que grâce à la simplicité et à l'élégance de Lisp et d'Algol cette inclusion est la solution la plus simple et la plus puissante pour incorporer à Algol la possibilité de traitement des listes ; ces dernières sont commodes pour représenter les arbres qui, nous le verrons plus tard, n'en sont qu'un cas particulier. Les "records" suggérés par HOARE [Ho] seront, quand l'on disposera des moyens nécessaires (temps et place en machine), une solution plus rationnelle que Lisp-Algol pour traiter les problèmes dont nous parlerons par la suite.

V.1. Procédures de base (première version)

Décrivons d'abord les opérandes manipulés par les procédures de base. Ces opérandes sont des listes qui doivent respecter les règles syntaxiques suivantes :

<liste> ::= <atome> | (<liste d'éléments>)
<liste d'éléments> ::= <liste> | <liste d'éléments> <blanc> <liste>
<atome> ::= ( ) | <élément atomique>
<élément atomique> ::= <symbole> | <symbole> <élément atomique>
<symbole> ::= tout symbole disponible sauf "(", ")", "␣"
<blanc> ::= ␣ | ␣ blanc

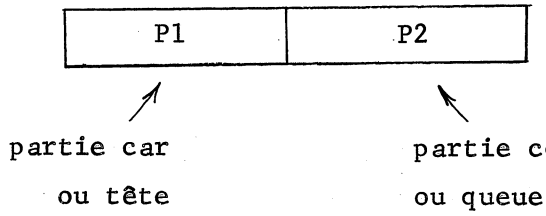
Remarquons que les blancs seront par la suite représentés par des espaces et que l'<atome> : ( ) est appelé nil. Signalons aussi que les règles ci-dessus imposent des parenthèses extérieures dans le cas où la liste possède plus d'un atome.

Exemple : Soit la liste :

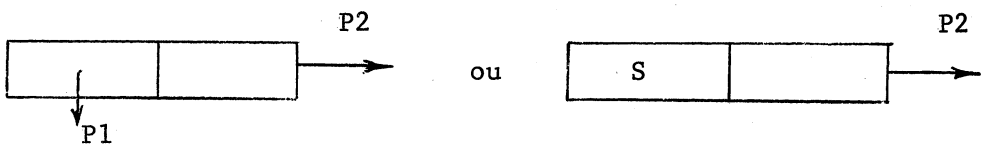
(A (BC D) ())

A, (BC D) et ( ) sont des éléments de cette liste ; A, BC,D et ( ) sont chacun un <atome> ; (BC D) est une <liste> que nous appellerons sous-liste.

La sémantique associée aux définitions syntaxiques données ci-dessus est étroitement liée à la façon dont les éléments d'une liste sont emmagasinés en mémoire. Il est plus facile d'expliquer graphiquement le concept de liste. Considérons un "mot-machine" représenté par une "boîte" partagée en deux parties :

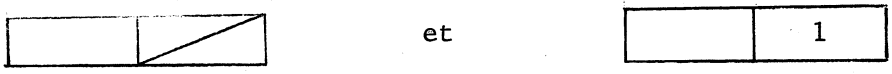


P1 et P2 sont les contenus de la partie gauche et droite du mot-machine et les parties elles-mêmes seront appelées car et cdr. P1 et P2 représentent en général une adresse mais P1 peut aussi parfois représenter un symbole. On distingue ces deux cas par les représentations suivantes :

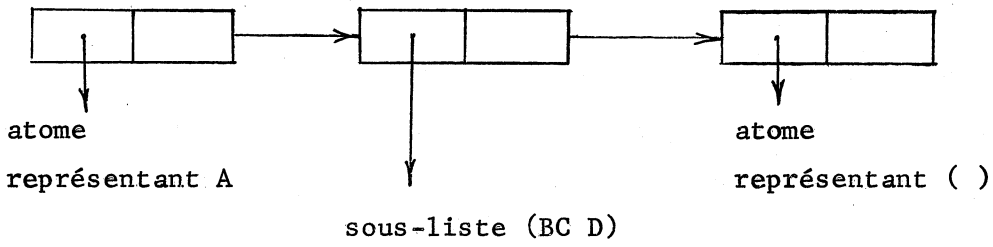


selon que car contient une adresse P1 ou un symbole S. Dans ce qui suit les adresses P1 et P2 sont des pointeurs (représentés par les flèches) indiquant où la prochaine information est emmaganisée : P2 pointe vers le prochain <élément> d'une <liste d'éléments>. P1 pointe soit vers une sous-liste soit vers un <atome>, c'est-à-dire que P1 décrit la nature d'un élément d'une liste d'éléments.

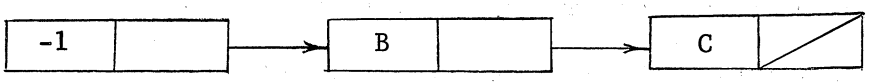
La parenthèse ")" qui indique la fin d'une liste ou d'une sous-liste sera représentée par une adresse spéciale en partie cdr. Cette adresse est, dans notre cas, celle de la mémoire 1 et nous considérons les deux représentations ci-dessous comme identiques :



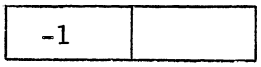
Le "squelette" de la liste donnée en exemple peut maintenant être schematisé par :



Bien que nous ayons déjà les moyens de représenter graphiquement la sous-liste (BC D), il nous faut encore expliquer la représentation graphique d'un atome. Les atomes sont rangés dans d'autres listes que nous appelons listes de propriété ("property lists" de Lisp). Les listes de propriété sont caractérisées par leur première mémoire dont la partie car contient la valeur -1. Par exemple la liste de propriété de l'atome BC est la suivante :



L'adresse de la première mémoire d'une liste de propriété est utilisée comme pointeur dans la partie car d'un élément du squelette de liste que nous avons déjà décrit. Remarquons que les contenus des parties car de la deuxième mémoire et des mémoires suivantes de la liste de propriété contiennent les symboles qui forment l'atome en question. Observons aussi qu'une partie cdr contenant la valeur 1 est également utilisée pour indiquer la fin d'une liste de propriété et que la liste de propriété de "()" (appelé NIL) est emmagasinée dans la mémoire d'adresse 1. Cette dernière est représentée par :



Comme nous décrivons les procédures de base en Algol, nous allons diviser la mémoire de travail en deux tableaux (de type entier) à une dimension que nous appellerons TCAR et TCDR. Le couple formé par un élément de TCAR et l'élément correspondant de TCDR joue d'ailleurs le même rôle que le mot-machine en Lisp pur. Remarquons que si la machine pour laquelle on écrit ces procédures n'a pas suffisamment de mémoires, il sera nécessaire d'écrire ces procédures en code et de partager le mot-machine comme en Lisp pur. Cependant, l'utilisation d'un mot-machine pour chaque partie car ou cdr présente un important avantage : elle permet l'emploi des mémoires secondaires de la façon suggérée en Appendice A.

Présentons les procédures de base avant de les décrire en Algol. En voici la liste :

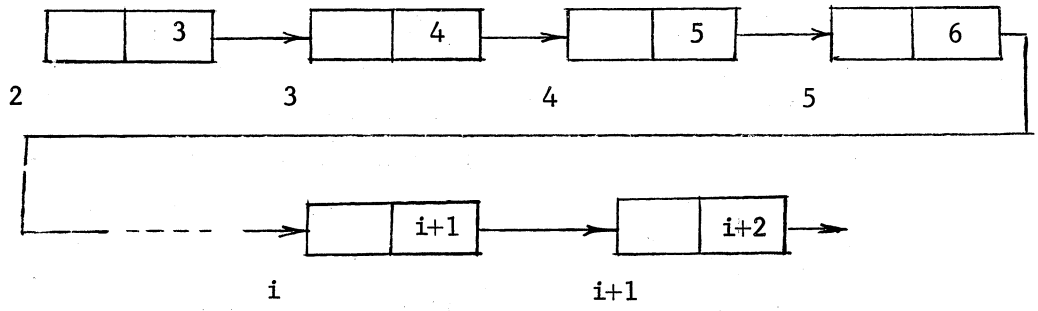
		BUT
1.	PREPARER	Préparation de la liste libre
2.	CAR	Détermination de la tête d'une liste
3.	CDR	Détermination de la queue d'une liste
4.	CONS	Construction d'une nouvelle liste
5.	ATOM	Prédicat d'atomicité
6.	EQ	Prédicat d'égalité
7.	LIRE LISTE	Lecture de listes
8.	NOMMER	Désignation de listes
9.	ECRIRE LISTE	Impression de listes
10.	GC1	Récupération des listes inutiles

LIRE LISTE, NOMMER et ECRIRE LISTE sont des procédures de transmission (entrée-sortie) ; CAR, CDR et CONS sont des procédures de manipulation de listes ; EQ et ATOM sont des prédicats : le premier permet la reconnaissance des symboles et le deuxième vérifie si un élément de liste est atomique. PREPARER et GC1 sont des procédures auxiliaires. Passons maintenant à la description de toutes ces procédures.

procédure PREPARER (<entier>)

PREPARER sert à initialiser une utilisation ultérieure des procédures Lisp. Elle a un paramètre qui doit être le même entier que la borne supérieure des tableaux TCAR et TCDR lors de leur déclaration. Le but principal de cette procédure est de préparer une liste "libre" qui contient toutes les mémoires qui étaient déclarées disponibles.

Chaque partie droite (TCDR) contient l'adresse de la prochaine mémoire disponible, qui initialement (c'est-à-dire après l'exécution de PREPARER) est égale à l'adresse de la mémoire suivante. Cette première action est représentée schématiquement de la façon suivante :



Remarquons que toutes les mémoires nécessaires à une opération donnée sont prises dans la liste libre et y sont remises quand elles sont libérées par une instruction appropriée du programmeur (voir GC1) ; nous verrons plus tard que l'adresse de la mémoire disponible suivante n'est pas, en général, l'adresse du mot suivant dans le mémoire de la machine.

La deuxième action de PREPARER est de construire la liste de propriété de NIL et d'initialiser le pointeur IPD qui indique l'adresse de la première mémoire de la liste libre. Ensuite PREPARER appelle les procédures LIRE et SAUTER pour initialiser l'utilisation ultérieure de LIRE LISTE. La description de LIRE et SAUTER sera donnée lors de la présentation de LIRE LISTE. L'initialisation des variables globales LA (liste d'atomes) et T à NIL est faite elle aussi par PREPARER. Leur but sera expliqué lors de la description des procédures LIRE LISTE et GC1.

entier procédure CAR (<expression entière représentant une liste>) ;

L'indicateur de fonction CAR est assez simple. Sa valeur est le contenu de TCAR [P], P étant le paramètre effectif représentant le pointeur d'une liste.

entier procédure CDR (<expression entière représentant une liste>) ;

Cet indicateur de fonction a comme valeur celle du pointeur rangé en partie cdr de P (P étant le paramètre effectif).

entier procédure CONS (<expression entière représentant une liste>,
<expression entière représentant une liste>) ;

Le but de cette procédure est de prendre un couple TCAR - TCDR de la liste libre et de ranger en TCAR la valeur du premier paramètre et dans TCDR la valeur du second. La valeur de l'indicateur de fonction CONS après l'appel est un entier qui représente le pointeur de la nouvelle liste qui vient d'être construite. Le cas particulier où le deuxième paramètre effectif représente un élément atomique demande l'utilisation d'une mémoire supplémentaire prise elle aussi dans la liste libre et liée à la précédente pour former l'ensemble voulu. Observons que la procédure CONS décrite ici est légèrement différente de la procédure CONS utilisée en Lisp. Cette différence a été introduite pour éviter la notation pointée ("dot") de Lisp.

booléen procédure ATOM (<expression entière représentant une liste>) ;

L'indicateur de fonction ATOM prend la valeur vrai si le paramètre effectif pointe vers une liste de propriété, c'est-à-dire si

TCAR [P] = -1, P étant le paramètre effectif. Dans les autres cas ATOM prend la valeur faux.

booléen procédure EQ (<expression entière représentant une liste>, <expression entière représentant une liste>) ;

Cet indicateur de fonction a pour valeur vrai si les deux listes considérées sont atomiques et ont des listes de propriété identiques ; autrement il prend la valeur faux.

Nous ne recommandons pas l'utilisation directe de la procédure EQ. Nous verrons qu'elle n'est appelée que dans le corps de la procédure LIRE LISTE, pour vérifier interprétativement l'égalité de deux atomes. Lorsque les atomes sont lus par LIRE LISTE, il n'existe en mémoire qu'un seul exemplaire de plusieurs atomes identiques ; leur comparaison peut alors se faire au moyen de l'opérateur binaire Algol "=" qui vérifie l'égalité des adresses des deux listes opérandes.

entier procédure LIRE LISTE ;

Cette procédure est utilisée pour :

a. Lire des données sur cartes et les ranger sous forme de liste.

b. Affecter à LIRE LISTE la valeur du pointeur du premier élément de la liste lue.

Pour l'opération décrite en a. il est nécessaire d'employer une procédure d'entrée capable de lire un symbole alphanumérique et de lui donner une valeur numérique correspondante de type entier. Les procédures LIRE SYMBOLE et ECRIRE SYMBOLE sont les seules à posséder un corps en code. Elles ont été conçues selon les recommandations du

groupe ALGOL IFIP [RIO] : la procédure INSYMBOL (Channel, String, Destination) est présentée sous la forme :

LIRE SYMBOLE (0, '␣ ( ) ABCDEF ... ' , TEMP) ;

cette instruction lit le prochain symbole sur carte et affecte à la variable entière TEMP la valeur numérique correspondant au numéro d'ordre de cet élément dans "String". Ainsi si une "(" est le premier caractère d'une carte donnée, après l'exécution de LIRE SYMBOLE, TEMP aura la valeur 2. Les blancs ont comme correspondant numérique la valeur 1.

Nous présentons ici deux versions de la procédure LIRE LISTE. La première, LIRE LISTE 1, a été écrite par F. GENUYS qui l'a définie au moyen d'appels récursifs par la procédure CONS. Avant de la décrire il convient de remarquer que la définition syntaxique des listes données précédemment est trop rigide en ce qui concerne les blancs et rendrait lourde la tâche de préparer les listes de données : il serait intéressant de la modifier légèrement pour permettre l'introduction de blancs facultatifs entre les parenthèses. On a tenu compte de cette modification dans la procédure LIRE LISTE qui utilise comme procédures auxiliaires LIRE, SAUTER, LIRE ATOME, LIRE FIN ATOME et LIRE SUITE. Les procédures LIRE et SAUTER, déjà mentionnées dans la description de PREPARER, ont pour but de placer dans TEMP (variable entière déclarée dans un bloc extérieur) le premier symbole non blanc de la chaîne d'entrée. La procédure LIRE lit un symbole et SAUTER réutilise LIRE si le symbole lu est un blanc ; sinon SAUTER ne produit aucun effet.

Le fonctionnement de LIRE LISTE peut être brièvement décrit de la façon suivante : TEMP doit contenir initialement ou bien la valeur 2 représentant une "(" ou bien une valeur plus grande que 3. Dans ce dernier cas la liste d'entrée est un atome et le contrôle est transféré à LIRE ATOME qui construit (en utilisant CONS) le début de la liste de propriété. Pour ranger les symboles successifs de l'atome dans la liste de propriété LIRE ATOME utilise LIRE FIN ATOME qui s'appelle elle-même jusqu'à la lecture d'un délimiteur de fin d'atome (blanc ou parenthèse).

La procédure LIRE ATOME vérifie si l'atome lu a déjà été créé précédemment. La variable entière LA (liste d'atomes) désigne une liste contenant tous les atomes existants en ce point du programme. Si l'atome actuellement considéré ne s'y trouve pas il y est rajouté. LIRE ATOME prend la valeur du pointeur désignant ce nouvel atome.

Revenons au cas où TEMP contenait la valeur 2 indiquant que le symbole lu était une "(". Dans ce cas LIRE LISTE lit le prochain symbole et si ce dernier est une ")" elle a pour valeur 1, c'est-à-dire le pointeur de la liste de propriété de NIL. Autrement elle appelle LIRE SUITE qui construit une liste ayant pour tête LIRE LISTE et pour queue LIRE SUITE, à moins qu'une ")" soit lue, indiquant la fin d'une liste ou sous-liste. Il est important de remarquer que la fin de LIRE LISTE utilise LIRE et SAUTER pour placer de nouveau

dans TEMP la valeur du prochain symbole non blanc ; cette action a pour but de permettre la vérification d'une fin de liste (voir partie cdr de CONS dans LIRE SUITE) et aussi de préparer TEMP pour la lecture d'une prochaine liste.

En tenant compte de la façon dont les procédures LIRE et SAUTER sont utilisées, il est toujours nécessaire d'avoir au moins un symbole non blanc supplémentaire (par exemple une ")") après la dernière des "listes données".

En résumé on peut dire que LIRE LISTE lit une liste, LIRE ATOME et LIRE FIN ATOME lisent un atome et LIRE SUITE lit l'intérieur d'une paire de parenthèses si son contenu n'est pas vide.

La deuxième version de LIRE LISTE, que nous avions proposée au départ, est moins élégante que la première, mais en revanche plus efficace. Nous la présentons ici pour illustrer la technique de tables d'états décrite au chapitre III. On suppose qu'avant l'appel de LIRE LISTE 2 la table d'états TE $[i,j]$ contient les valeurs suivantes (mises en place, par exemple, lors de l'appel de la procédure PREPARER) :

i \ j	␣	(	)	<symbole>	commentaire
1	1	3	12	5	état initial
2	2	3	7	5	après un blanc
3	4	3	8	5	après une (
4	4	3	8	5	après (␣
5	6	11	10	9	après le premier <symbole
6	2	3	7	5	après un <atome>
7	2	3	7	5	après une)
8	2	3	7	5	après ( )
9	6	11	10	9	après le deuxième <symbole
10	2	3	7	5	après <symbole>)
11	4	3	8	5	après <symbole (

Remarquons qu'un automate d'états finis suffit pour reconnaître la séquence des blancs et des atomes. Cependant il est nécessaire d'utiliser une pile pour le comptage des parenthèses ; autrement dit la structure de liste présentée est "context-free" et il faut pour l'analyser un automate à pile. La manipulation de cette pile est déclenchée par certains états de l'automate qui permettent d'empiler ou d'enlever les éléments contenus dans la pile.

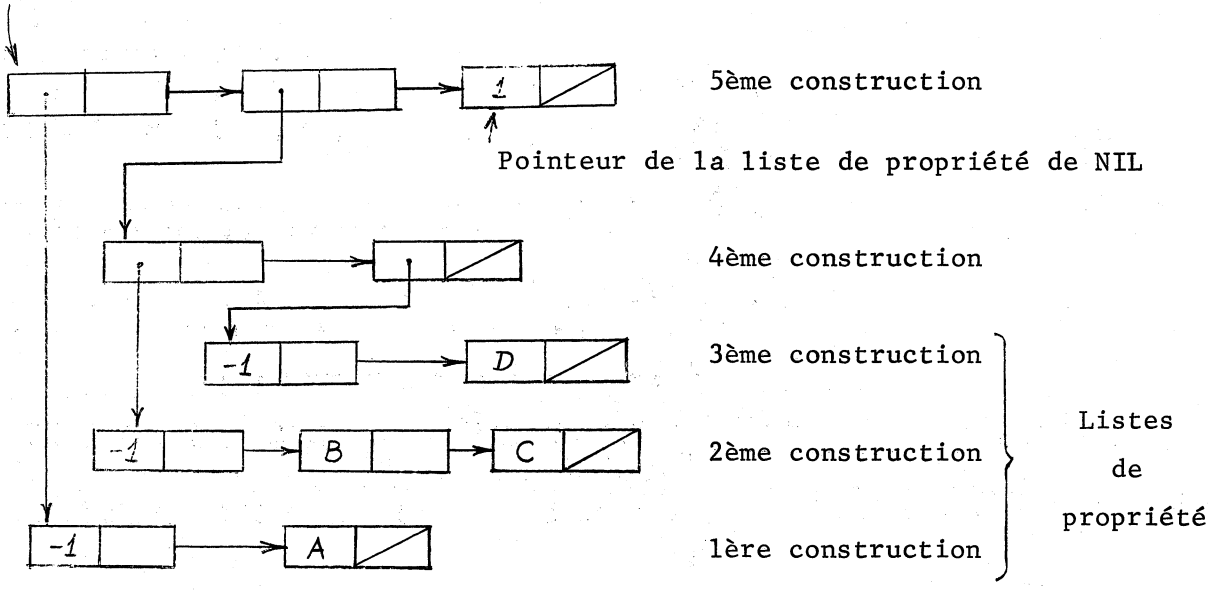
Signalons finalement que l'utilisation d'une pile auxiliaire est semblable à celle employée lors de la transformation d'une expression arithmétique complètement parenthésée en chaîne polonaise : les parenthèses ouvrantes sont mises directement dans la pile ; lors de la lecture d'un atome, l'adresse de sa liste de propriété y est également mise. C'est la lecture du symbole ")" qui déclenche

l'élimination des éléments de la pile jusqu'à la première "(" et le remplacement de ce dernier par l'adresse de la sous-liste correspondante.

Exemple : L'appel de LIRE LISTE 1 (ou LIRE LISTE 2) pour lire la liste :

(A (BC D) ())

provoque le rangement suivant en mémoire :



entier procédure NOMMER (<chaîne>) ;

Cette procédure est semblable à LIRE LISTE : alors que LIRE LISTE permet de ranger des listes définies sur cartes (ou bandes), la procédure NOMMER permet de ranger des listes définies à l'intérieur d'un programme. NOMMER diffère de LIRE LISTE par l'utilisation :

- d'une zone de rangement provisoire dans laquelle est écrit le paramètre effectif <chaîne> ;

- d'un canal spécial de lecture pour la procédure LIRE SYMBOLE.

Un appel de cet indicateur de fonction affecte à l'identificateur NOMMER la valeur du pointeur de la liste représentée par chaîne . On peut remarquer que l'effet de NOMMER est analogue à celui de QUOTE en Lisp. Nous n'avons pas listé la procédure NOMMER en fin de paragraphe, étant donné qu'elle peut être facilement reconstituée par l'utilisation de LIRE LISTE.

entier procédure ECRIRE LISTE (expression entière représentant une liste) ;

Un appel de cette procédure qui a comme paramètre la variable entière représentant une liste construite au moyen des procédures Lisp, permet d'imprimer la liste correspondante. L'impression est faite de telle manière qu'il n'y a aucune modification à faire pour la lire ultérieurement au moyen de LIRE LISTE.

Remarquons que ECRIRE LISTE est un indicateur de fonction du type entier dont la valeur après exécution est celle du pointeur de la liste à imprimer. Ceci permet d'utiliser la procédure ECRIRE LISTE comme paramètre effectif d'autres procédures. Cela peut aussi faciliter la détection des erreurs quand on veut introduire des impressions intermédiaires.

procédure GC1 (expression entière représentant une liste ,
<étiquette>) ;

La procédure GC1* sert à récupérer les listes inutiles. Nous verrons dans les exemples du paragraphe V.5 que les transformations

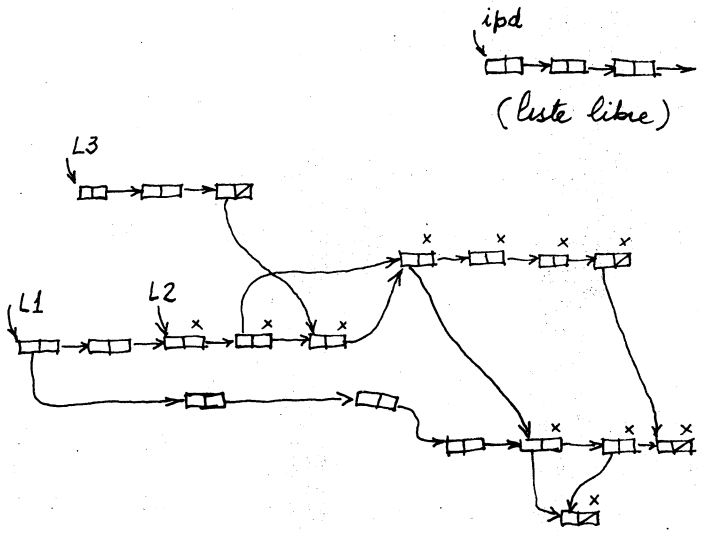
---

(*) GC est un abréviation pour "garbage collection", expression consacrée par Mc CARTHY et signifiant récupération des listes inutiles.

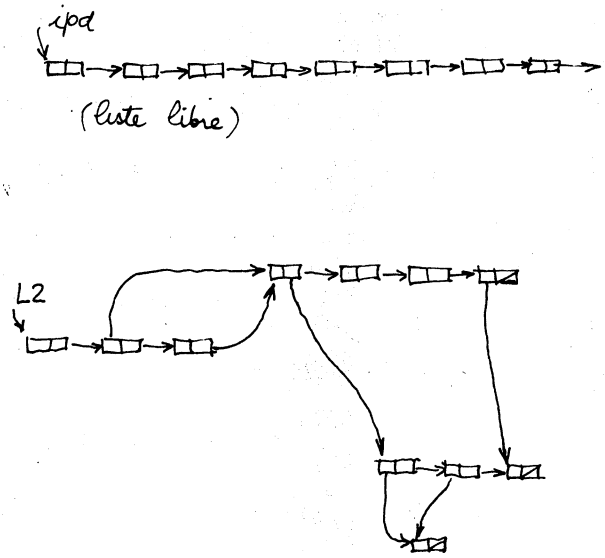
des listes sont presque toujours faites au moyen de nouvelles constructions, c'est-à-dire par des appels à la fonction CONS. Les listes intermédiaires utilisées dans la construction d'une liste résultante sont souvent abandonnées, et il devient indispensable de récupérer l'espace qu'elles utilisent, compte tenu de la nature des problèmes réels et de la capacité des mémoires rapides des ordinateurs. L'expérience montre que pour certains problèmes le temps total de récupération est supérieur au temps de la résolution du problème lui-même. Ceci montre l'importance des méthodes de récupération efficace dans les langages de listes en général. SCHORR et WAITE donnent en [SW] une description globale des méthodes existantes.

Le mécanisme utilisé par GC1 est celui qu'a suggéré Mc CARTHY pour le compilateur Lisp. Cette procédure efface toutes les listes existant en mémoire sauf celles dont les pointeurs ont été préalablement ajoutés à la liste T donnée comme premier paramètre. Dans une phase initiale toutes les listes nommées dans T sont marquées. Le marquage se fait par la procédure "marquer" et consiste à inverser le signe de la partie cdr d'un mot. Un balayage linéaire de la mémoire disponible permet ensuite d'enlever les marques des mots qui en ont une et d'ajouter à la liste libre les mots non marqués. La sortie vers l'<étiquette> se fait lorsqu'aucun mot ne peut être récupéré. "n max" indique la borne supérieure de TCAR et TCDR et "ipd" indique la première mémoire disponible dans la liste libre. La Figure 1 montre de façon imagée le fonctionnement de GC1.

Dans les pages suivantes le lecteur trouvera en Algol un listage des procédures Lisp.



Configuration de la mémoire avant la récupération



Configuration après la récupération

Fonctionnement du récupérateur GC1 quand on lui demande d'effacer toutes les listes sauf L2

Figure 1

```
procédure PREPARER (N) ; valeur N ; entier N ;
    début entier I ;
       commentaire préparation de la liste libre ;
       pour I := 2 pas 1 jusqua N faire
          début
                TCAR [I] := 0 ;
                TCDR [I] := I + 1
          fin ;

       commentaire construction de la liste de propriété de NIL ;
       TCAR [1] := -1 ;
       TCDR [1] := 1 ;
       IPD := 2 ;

       commentaire lecture du premier symbole non blanc d'une liste,
                   pour initialiser l'appel de LIRE LISTE ;
       LIRE ;
       SAUTER ;
       LA := T := 1
    fin procédure PREPARER ;


entier procédure CAR (A) ; valeur A ; entier A ;
     CAR := TCAR [A] ;


entier procédure CDR (A) ; valeur A ; entier A ;
     CDR := TCDR [A] ;
```

```
entier procédure CONS (A,B) ; valeur A,B ; entier A,B ;
    début entier I ;
        CONS := I := IPD ;
        IPD := TCDR [IPD] ;
        TCAR [I] := A ;
        si TCAR [B] = -1 ∧ B ≠ 1
           alors début
                    I := IPD ;
                    IPD := TCDR [IPD] ;
                    TCAR [I] := B ;
                    TCDR [I] := 1
                 fin
           sinon TCDR [I] := B
    fin procédure CONS ;

booléen procédure ATOM (A) ; valeur A ; entier A ;
        ATOM := TCAR [A] = -1 ;

booléen procédure EQ (A,B) ; valeur A,B ; entier A,B ;
    début
        si TCAR [A] ≠ -1 ∨ TCAR [B] ≠ -1 alors allera FAUX ;
        si A = B alors allera VRAI ;
        BOUCLE :
        A := TCDR [A] ; B := TCDR [B] ;
        allera si A = 1 ∧ B = 1 alors VRAI sinon
                  si TCAR [A] ≠ TCDR [B] alors FAUX sinon BOUCLE ;
        FAUX : EQ := faux ; allera FINI ;
        VRAI : EQ := vrai ;
        FINI :
    fin procédure EQ ;
```

```
entier procédure LIRE LISTE 1 ;

    si TEMP > 3 alors LIRE LISTE := LIRE ATOME
                sinon début
                        LIRE ;
                        SAUTER ;
                        LIRE LISTE 1 := si TEMP = 3 alors 1 sinon LIRE SUITE ;
                        LIRE ;
                        SAUTER
                      fin procédure LIRE LISTE ;


procédure LIRE ;

    LIRE SYMBOLE (0, '␣()ABCDE...0123...*/+-', TEMP) ;


procédure SAUTER ;

    A : si TEMP = 1 alors début
                            LIRE ;
                            allera A
                          fin procédure SAUTER ;
```

```
entier procédure LIRE ATOME ;
    début entier t1, t2 ;
        t1 := CONS (-1, LIRE FIN ATOME) ;
        SAUTER ;
        t2 := RECHERCHE (t1, LA) ;
        si t2=1 alors
                    début
                        LIRE ATOME := t1 ;
                        LA := CONS (t1, LA)
                    fin
                sinon
                    début
                        LIRE ATOME := t2 ;
                        LIBERER ATOME (t1)
                    fin
    fin procédure LIRE ATOME ;


entier procédure RECHERCHE (t, L) ; valeur t, L ; entier t, L ;
    RECHERCHE := si L=1 alors 1 sinon
                 si EQ (CAR(L), t) alors CAR(L)
                                   sinon RECHERCHE (t, CDR(L)) ;


entier procédure LIRE FIN ATOME ;
    début entier T ;
        T := TEMP ;
        LIRE ;
        LIRE FIN ATOME := CONS (T, si TEMP ⩽ 3 alors 1 sinon LIRE FIN ATOME)
    fin procédure LIRE FIN ATOME ;
```

```
entier procédure LIRE SUITE ;
     LIRE SUITE := CONS (LIRE LISTE, si TEMP = 3 alors 1 sinon LIRE SUITE) ;


procédure LIBERER ATOME (V) ; valeur V ; entier V ;
     début entier T ;
          pour T := V tant que T ≠ 1 faire
              début
                  V := TCDR [T] ;
                  LIBERER MOT (T)
              fin
     fin procédure LIBERER ATOME ;


procédure LIBERER MOT (U) ; valeur U ; entier U ;
          début
              TCAR [U] := 0 ;
              TCDR [U] := IPD ;
              IPD := U
          fin procédure LIBERER MOT ;


entier procédure LIRE LISTE 2 ;
     début entier tableau PILE [0:99] ;
          entier TEMP, I, K, IP, ETAT, OUV PARENTHESE ;
          aiguillage ACTION := E1, E2, E3, E4, E5, E6,
                               E7, E8, E9, E10, ERREUR ;
```

```
OUV PARENTHESE := -2 ;

IP := 0 ;

ETAT := 1 ;

RETOUR :
LIRE SYMBOLE (0, ␣() ABC ... 012 ... 9*/+-' , TEMP) ;
ETAT := TE [ ETAT, si TEMP>3 alors 4 sinon TEMP] ;
allera ACTION [ETAT] ;

commentaire lecture d'une "(" ;
E3 :
PILE [IP] := OUV PARENTHESE ;
IP := IP+1 ;

commentaire lecture d'un blanc ;
E1 : E2 : E4 :
allera RETOUR ;

commentaire lecture du premier symbole d'un atome ;
E5 :
TCAR [IPD] := -1 ;
PILE [IP] := IPD ;
IP := IP+1 ;

commentaire lecture d'un symbole ;
E9 :
IPD := TCDR [IPD] ;
TCAR [IPD] := TEMP ;
allera RETOUR ;

commentaire lecture d'un "()" ;
E8 :
PILE [IP-1] := 1 ;
allera VERIFIER FIN DE LECTURE ;
```

```
ERREUR :
ECRIRE ('erreur lecture') ;
allera FINALE ;

commentaire fin de lecture d'un atome ;
E6 : E11 : E10 :
TEMP := TCDR [IPD] ;
TCDR [IPD] := 1 ;
IPD := TEMP ;
TEMP := RECHERCHE (PILE [IP-1], LA) ;
si TEMP = 1 alors LA := CONS (PILE [IP-1], LA)
            sinon
                début
                    LIBERER ATOME (PILE [IP-1]) ;
                    PILE [IP-1] := TEMP
                fin ;
si ETAT = 6 alors allera VERIFIER FIN DE LECTURE
            sinon si ETAT = 11 alors allera E3 ;

commentaire lecture d'une ")" ;
E7 :
TEMP := IPD ;
pour I := IP -1 pas -1 jusqua 0 faire
     si PILE [I] = OUV PARENTHESE alors allera SORTIE DE POUR ;

SORTIE DE POUR :
K := I ;
pour I := I+1 tantque I ≠ IP-1 faire
      début
          TCAR [IPD] := PILE [I] ;
          IPD := TCDR [IPD]
      fin ;
```

```
            TCAR [IPD] := PILE [IP-1] ;
            I := TCDR [IPD] ;
            TCDR [IPD] := 1 ;
            IPD := I ;
            PILE [K] := TEMP ;
            IP := K+1 ;

            VERIFIER FIN DE LECTURE :
            si IP ≠ 1 alors allera RETOUR ;

            LIRE LISTE 2 := PILE [0] ;

            FINALE :

        fin procédure LIRE LISTE ;


entier procédure ECRIRE LISTE (V) ; valeur V ; entier V ;
        début
            ECRIRE LISTE := V ;
            ECRIRE LISTE MODIF (V)
        fin procédure ECRIRE LISTE ;


procédure ECRIRE LISTE MODIF (V) ; valeur V ; entier V ;
        si ATOM (V) alors ECRIRE ATOME (V)
                    sinon début
                            ECRIRE SYMBOLE (2, '(', 1) ;
                            ECRIRE SUITE (V)
                          fin procédure ECRIRE LISTE MODIF ;
```

```
procédure ECRIRE SUITE (V) ; valeur V ; entier V ;
    début
        ECRIRE LISTE MODIF (CAR (V)) ;
        si CDR (V) = 1 alors ECRIRE SYMBOLE (2, ')', 1)
                       sinon ECRIRE LISTE MODIF (CDR (V))
    fin procédure ECRIRE SUITE ;


procédure ECRIRE ATOME (V) ; valeur V ; entier V ;
    début
        si V = 1 alors début
                           ECRIRE SYMBOLE (2, '(', 1) ;
                           ECRIRE SYMBOLE (2, ')', 1)
                       fin
                 sinon
        pour V := TCDR [V] tantque V ≠ 1 faire
            ECRIRE SYMBOLE (2, '␣() ABC ... ' , TCAR [V]) ;
        ECRIRE SYMBOLE (2, '␣' , 1)
    fin procédure ECRIRE ATOME ;


procédure GC1  (T, épuisé) ; valeur T ; entier T ; étiquette épuisé ;
    début entier i, temp ;
        temp := IPD ;
        marquer (T) ;
        pour i := 2 pas 1 jusqua nmax faire
            si CDR (i) < 0 alors TCDR [i] := -TCDR [i]
                           sinon début
                                     TCAR [i] := 0 ;
                                     TCDR [i] := IPD ;
                                     IPD := i
                                 fin ;
        si temp = IPD alors allera épuisé
    fin GC1 ;
```

```
procédure marquer (T) ; valeur T ; entier T ;
    si T≠1 alors début
                    marquer liste (CAR (T)) ;
                    marquer (CDR (T))
                fin marquer ;

procédure marquer liste (L) ; valeur L ; entier L ;
    si atome (L) alors marquer atome (L)
                 sinon début
                           marquer liste (CAR (L)) ;
                           si CDR (L) > 0 alors
                               début
                                   marquer liste (CDR (L)) ;
                                   TCDR [L] := -TCDR [L]
                               fin
                       fin marquer liste ;

procédure marquer atome (A) ; valeur A ; entier A ;
    début entier t ;
        si A ≠ 1 alors
            début
                E : t := CDR (A)
                   si t > 0 alors
                         début
                             TCDR [A] := -TCDR [A] ;
                             si t ≠ 1 alors
                                  début
                                      A := t ;
                                      allera E
                                  fin
                         fin
            fin
    fin marquer atome ;
```

V.2. Deuxième version

Afin de montrer l'importance de la récupération dans un langage de listes, nous proposons une deuxième version des procédures que nous venons de décrire. La position de certains programmeurs qui affirment qu'il faut choisir la méthode de récupération avant de passer à la réalisation du compilateur d'un langage de listes n'est en effet pas déraisonnable (cf. [BB]). L'emploi de la procédure GC2, décrite en fin de ce paragraphe, élimine le besoin de la liste libre utilisée dans la première version. Après l'exécution de GC2, les listes utiles sont tassées dans la partie inférieure des tableaux TCAR et TCDR, et les mots consécutifs dans la partie supérieure sont disponibles pour les constructions ultérieures. Passons donc à la description de GC2, dont le mécanisme a été suggéré en [BB].

GC2 efface toutes les listes en mémoire, sauf celles dont les pointeurs ont été ajoutés à la liste T. Dans une première phase, toutes les listes appartenant à T sont marquées. Les deux indices i et j pointent initialement vers les deux extrémités des tableaux TCAR et TCDR.

L'indice i pointe vers la partie inférieure, et on le fait avancer jusqu'à trouver un mot non marqué. L'indice j progresse d'une façon semblable, mais vers le bas, pour chercher un mot marqué. Quand on trouve ce dernier, on range son contenu dans la mémoire repérée par i, et on range la valeur de i dans le mot pointé par j. On répète

ce processus jusqu'à ce que i et j se rencontrent à une adresse m. On fait ensuite un dernier balayage pour rectifier les références aux adresses plus grandes que m, en les remplaçant par les contenus des mots vers lesquels elles pointent. Un balayage similaire est nécessaire pour rectifier les pointeurs rangés dans T. La signification de n max, ipd et épuisée est la même que dans GC1.

Cette deuxième version comporte dans les procédures PREPARER et CONS des changements évidents que nous jugeons inutiles de préciser ici.

Compte tenu du travail supplémentaire de retassement, le temps d'exécution de GC2 est évidemment supérieur à celui de GC1 et la différence entre ces temps d'exécution augmente avec la taille de la mémoire réservée aux listes.

Dans le cas de calculateurs n'ayant qu'une mémoire (rapide), il n'est pas particulièrement avantageux d'utiliser GC2, bien que ce dernier utilise la mémoire d'une façon plus rationnelle dans le cas où la profondeur des listes est importante et a priori inconnue.

La comparaison entre GC1 et GC2 devient plus difficile dans le cas où les deux mémoires (rapide et lente) sont utilisées de la façon décrite en Appendice A. L'étude comparative de GC1 et GC2 lorsqu'on utilise des mémoires secondaires est faite en Appendice B.

```
procédure GC2 (T, épuisé) ; valeur T ; entier T ; étiquette épuisé ;
    début entier i, j ;
        marquer (T) ;
        i := 1 ; j := n max + 1 ;
    A : i := i + 1 ;
        si CDR (i) < 0 alors
            début
                TCDR [i] := -TCDR [i] ;
                allera si i ≠ nmax alors A sinon épuisé
            fin ;
    B : si i < j alors
            début
                u := j - 1 ;
                si CDR (j) > 0 alors allera B ;
                TCAR [i] := TCAR [j] ; TCDR [i] := TCDR [j] ;
                TCAR [j] := i ; TCDR [j] := 0 ;
                allera A
            fin ;
        pour i := 1 pas 1 jusqua j faire
            début
                si CAR (i) > j alors TCAR [i] := TCAR [TCAR  i]  ;
                si CDR (i) > j alors TCDR [i] := TCAR [TCDR  i
            fin ;
         rectifier (T) ;
fin GC2 ;
procédure rectifier (T), valeur T ; entier T ;
            si T ≠ 1 alors début
                                si CAR (T) > j alors TCAR [T] := TCAR [CAR(T)] ;
                                rectifier (CDR (T))
                                fin rectifier ;
```

V.3. Remarques sur l'utilisation des procédures

Les procédures que nous venons de présenter sont utilisées dans des programmes Algol dont la structure est la suivante :

```
début entier N ;
      N := EDONNEE ;
  début entier tableau TCAR, TCDR [1:N] ;
       entier <variables globales de Lisp-Algol ;
  procédure PREPARER (K) ;
       <corps de la procédure> ;
  entier procedure CAR(L) ;
       <corps de la procédure> ;
                .
                .
                .
       <autres procédures Lisp-Algol  ;
                .
                .
                .
  entier <variables définies par l'utilisateur> ;
        <procédures définies par l'utilisateur> ;
  PROGRAMME PRINCIPAL :
  PREPARER (N) ;
   instructions définies par l'utilisateur>

  fin

fin ;
```

D'autres procédures utiles peuvent être facilement construites à partir des procédures de base. Par exemple la fonction "cadr" de Lisp est décrite en Lisp-Algol par la procédure :

```
entier procédure CADR (L) ; valeur L ; entier L ;
    CADR := CAR (CDR (L)) ;
```

On trouvera en [CD, Nhd] les procédures correspondant aux fonctions Lisp les plus couramment utilisées.

Résumons brièvement les avantages et les inconvénients de cette inclusion de Lisp dans Algol. Parmi les avantages mentionnons les suivants :

a. Les programmes Lisp-Algol sont plus lisibles que les S-expressions de Lisp pur. Ceci est surtout dû à l'existence en Algol des instructions d'affectation, des instructions allera, des aiguillages, etc.

b. L'évaluation numérique des expressions, lourde en Lisp pur, est immédiate en Algol.

c. Les programmes Lisp-Algol sont exécutés d'une façon moins interprétative qu'un programme en Lisp pur évalué par EVALQUOTE (cf. [Mc 2] et § V.5.2) et sont donc plus rapides.

d. L'inclusion permet l'utilisation des mémoires secondaires de la façon décrite en Appendice A.

Les inconvénients sont les suivants :

a. L'élimination de la notation pointée de Lisp ("dot") ne doit pas être oubliée par l'utilisateur de Lisp-Algol. Par exemple, si A pointe vers la liste (M N), B vers (P) et C vers l'atome P, le résultat de cons (A,B) est le même que celui de cons (A,C), c'est-à-dire la liste ((M N) P).

Remarquons en passant que l'introduction de la notation pointée de Lisp pur ne pose pas de grands problèmes.

b. La liste de propriété des atomes ne peut pas contenir de valeurs de type réel ; ceci est une conséquence de l'impossibilité en Algol de ranger un nombre réel dans un tableau entier. Cependant cet inconvénient peut être contourné par l'adoption de listes atomiques spéciales du type :

où p est un pointeur vers un tableau réel où l'on range les nombres en question. Les procédures LIRE et ECRIRE ATOME peuvent facilement être modifiées pour reconnaître cette nouvelle classe d'atomes. Remarquons que la récupération doit dans ce cas se prolonger jusqu'aux éléments du tableau réel ; ces derniers doivent alors être enchaînés pour constituer une liste libre, à moins que l'on ne fasse un retassement !

c. Il est impossible de déclencher automatiquement la récupération dans le corps de la procédure CONS, et c'est l'inconvénient qui nous chagrine le plus. Expliquons-nous au moyen d'un exemple : dans l'appel

CONS (CONS (A,B), CONS (C,D)) ,

si le déclenchement de GC1 (T, épuisé) se produit automatiquement à l'appel de CONS (C,D) le pointeur CONS (A,B) ne peut pas être ajouté à la liste T, puisque seul le compilateur Algol sait où il l'a mis dans la pile de récursivité. Ainsi, à moins de modifier le compilateur Algol ou d'introduire des procédures en code, il est impossible de faire une récupération

automatique des listes inutiles. Afin de rester en Algol pur, nous suggérons à l'utilisateur d'émailler son programme d'instructions du type :

si nd = pourcentage ∠ N alors GC (T, épuisé) ;

où N est la longueur de TCAR et TCDR et nd est le nombre de mémoires disponibles, à rechercher à chaque appel de CONS et à mettre à jour par GC. Le nombre réel "pourcentage" peut être choisi par l'utilisateur après quelques essais de son programme.

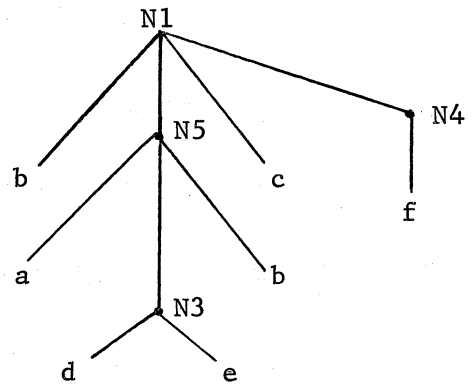
V.4. Représentation des arbres

Nous avons déjà indiqué qu'on peut commodément représenter les arbres sous forme de listes. Considérons la définition syntaxique suivante :

⟨arbre⟩ ::= (⟨noeud⟩ ⟨suite de branches⟩)
⟨suite de branches⟩ ::= ⟨branche⟩ | ⟨branche⟩ ⟨suite de branches⟩
⟨branche⟩ ::= ⟨feuille⟩ | ⟨arbre⟩
⟨feuille⟩ ::= ⟨élément atomique⟩
⟨noeud⟩ ::= ⟨élément atomique⟩

La comparaison avec la définition syntaxique de ⟨liste⟩ donnée en page 121 nous montre qu'un ⟨arbre⟩ n'est qu'un cas particulier d'une liste⟩. Plus précisément, un arbre est une liste dont le premier élément, toujours atomique, représente un ⟨noeud⟩ tandis que les éléments suivants, en nombre variable, sont soit des ⟨feuilles⟩ (noeuds terminaux) soit des ⟨arbres⟩ (noeuds non terminaux).

Exemple : L'arbre suivant :



est représenté par la liste :

(N1 b (N5 a (N3 d e) b) c (N4 f))

Nous avons évidemment choisi arbitrairement la convention suivante : le noeud correspond toujours au premier élément d'une liste ou sous-liste. Ceci a cependant sa raison d'être : le premier élément d'une liste est facilement repérable par la fonction CAR. Nous conviendrons aussi d'ordonner les branches issues d'un noeud suivant l'ordre d'apparition de gauche à droite des noeuds y aboutissant. Ainsi, la branche vers "a" est la première des branches sortant de N5, celle vers N3 la deuxième, etc.

V.5. Exemples

V.5.1. Algorithme d'analyse descendante

Nous présentons ici en Lisp-Algol l'algorithme d'analyse descendante déjà décrit au paragraphe IV.2.1. La procédure ANALYSE a comme paramètres les deux listes (piles) A et B. Les règles de l'automate sont elles aussi rangées sous forme de listes dont les pointeurs sont emmagasinés dans les tableaux RAG, RBG, RAD, RBD. La i-ème règle s'écrit de la façon suivante :

$$RAG[i] \quad , \quad RBG[i] \longrightarrow RAD[i] \quad , \quad RBD[i]$$

ce qui signifie : lorsqu'on trouve au sommet de la pile A (c'est-à-dire au début de la liste A) une configuration identique à celle de la liste RAG $[i]$ et lorsqu'on trouve au sommet de B une configuration identique à celle de RBG $[i]$, ces deux sommets sont respectivement remplacés par les listes rangées en RAD $[i]$ et RBD $[i]$. Afin de simuler séquentiellement le fonctionnement d'un automate non déterministe nous utilisons la récursivité en gardant l'état des piles A et B dans les variables locales AS et BS. La variable locale n garde la dernière information sur l'état de l'automate, c'est-à-dire le numéro de la dernière règle applicable.

La procédure COMPARAISON compare les sommets de A ou B avec une règle. Si le début de A ou B coïncide avec la règle rangée dans RAG ou RBG, le résultat de COMPARAISON est le résidu non coïncident

de A ou B. Sinon la valeur de COMPARAISON est NIL, c'est-à-dire 1. Le résultat de l'analyse est gardé dans le tableau TANALYSE : lorsqu'une règle de l'automate est applicable, son numéro est rangé dans ce tableau. La procédure booléenne SHAPER sert à voir s'il existe plus d'éléments dans la pile B que dans la pile A. Dans le cas affirmatif, les règles utilisées ont conduit à un impasse ; un retour en arrière est alors indispensable. L'analyse se termine lorsque les deux listes A et B contiennent uniquement l'élément spécial # . La procédure ARBRE permet alors de construire l'arbre d'analyse ; elle utilise uniquement TANALYSE et les règles de l'automate. Les autres procédures utilisées par ANALYSE ne nécessitent pas d'explications supplémentaires.

Montrons un exemple d'utilisation de cette procédure. La grammaire :

S ⟶ A B
A ⟶ a A
A ⟶ a
B ⟶ a B
B ⟶ a

est représentée de la façon suivante :

i	RAG [i]	RBG [i]	RAD [i]	RBD [i]
1	()	(S)	()	(A B)
2	()	(A)	()	(a A)
3	()	(A)	()	(a)
4	()	(B)	()	(a B)
5	()	(B)	()	(a)
6	(a)	(a)	()	()

Si A pointe vers la liste (a a a a #) et B vers (S #) la première analyse correspond aux éléments suivants rangés dans TANALYSE :

k	TANALYSE [k]
1	1
2	2
3	6
4	2
5	6
6	3
7	6
8	5
9	6

Ceci permet à la procédure ARBRE d'imprimer :

(S (A a (A a (A a))) (B a))

Les autres analyses obtenues par la suite sont :

(S (A a (A a)) (B a (B a)))

et

(S (A a) (B a (B a (B a))))

Deux remarques finales s'imposent. Signalons d'abord que, malgré son élégance et sa concision, la procédure ANALYSE n'est guère utilisable dans un compilateur dirigé par la syntaxe. D'un autre côté, cette procédure est générale et peut être utilisée avec les autres algorithmes d'analyse cités en [GP]. Cependant la procédure ARBRE doit être ré-écrite pour chaque cas particulier.

```
procédure ANALYSE (A, B) ; valeur A, B ; entier A, B ;
    début entier n, AS, BS ;
        n := 0 ;
  boucle : n := n+1 ;
           AS := COMPARAISON (A, RAG [n]) ;
           BS := COMPARAISON (B, RBG [n]) ;
           si AS ≠ 1 ∧ BS ≠ 1 alors
             début
                 k := k+1 ;
                 TANALYSE [k] := n ;
                 AS := APPEND (RAD [n], AS) ;
                 BS := APPEND (RBD [n], BS) ;
                 si LONGUEUR (AS) = 1 ∧ LONGUEUR (BS) = 1
                    alors ARBRE
                    sinon
                 si ¬ SHAPER (AS, BS) alors ANALYSE (AS, BS) ;
                  k := k-1
             fin ;
          si n = n final alors allera boucle
    fin ANALYSE ;

entier procédure COMPARAISON (U,V) ; valeur U,V ; entier U,V ;
    COMPARAISON := si V=1 alors U sinon
                     si CAR (U) ≠ CAR (V) alors 1 sinon
                        COMPARAISON (CDR (U), CDR (V)) ;

entier procédure APPEND (X,Y) ; valeur X,Y ; entier X,Y ;
    APPEND := si X=1 alors Y sinon CONS (CAR (X), APPEND (CDR (X), Y)) ;
```

```
entier procedure LONGUEUR (L) ; valeur L ; entier L ;
       LONGUEUR := si CDR (L) = 1 alors 1 sinon 1 + LONGUEUR (CDR (L)) ;

booleen procedure SHAPER (A,B) ; valeur A,B ; entier A,B ;
       SHAPER := LONGUEUR (A) < LONGUEUR (B) ;

procedure ARBRE ;
     début entier i ;
           entier procédure ARBRE 1 ;
                début entier t ;
                     t := TANALYSE [i] ;
                     i := i + 1 ;
                     ARBRE 1 := si RAG [t] ≠ 1 alors CAR (RBG [t])
                                                sinon
                                                CONS (CAR(RBG [t]), APPLIC(RBD[t]))
                fin ARBRE 1 ;
           entier procedure APPLIC (L) ; valeur L ; entier L ;
                     APPLIC := si L=1 alors 1
                                        sinon CONS (ARBRE 1, APPLIC (CDR (L))) ;
           i := 1 ;
           ECRIRE LISTE (ARBRE 1)
     fin ARBRE ;
```

V.5.2. Interprète de Lisp pur

Nous présentons ici la procédure EVALQUOTE qui évalue (interprète) un programme en Lisp pur écrit sous forme de S-expressions. Cette procédure a été mise au point avec la collaboration de T. EVANS. Une description plus ou moins détaillée d'EVALQUOTE se trouve en [Mc 2, BB]. De petites modifications ont été faites pour tenir compte de l'absence de la notation pointée. D'autres modifications ont été faites pour remplacer les procédures booléennes ATOM et EQ par les procédures entières ATOMM et EQM qui ont pour valeur TRUE et NIL au lieu de vrai ou faux.

L'intérêt de la procédure EVALQUOTE en Lisp-Algol est qu'elle devient, une fois compilée, totalement indépendante d'Algol. Son principal inconvénient est l'absence d'un récupérateur automatique. Pour tourner cette difficulté on peut soit employer les mémoires secondaires (cf. Appendice A), soit suivre la suggestion du paragraphe V.3, ou même employer ces deux moyens à la fois.

EVALQUOTE, vue comme un compilateur (interprète), contient elle aussi les deux phases de reconnaissance et d'action que nous avons mentionnées au chapitre II. Cependant la phase de reconnaissance est réduite à de simples vérifications du type :

```
si FN = SCAR alors ...
           sinon ...
si FN = SCDR alors ...
           sinon ...
etc.
```

Les vérifications combinées à la récursivité permettent une analyse de la syntaxe assez primitive du langage Lisp.

Sujet de recherche

Un autre programme d'intérêt pratique est celui qui traite le problème inverse : étant donnée une S-expression de Lisp pur, la traduire en un programme Lisp-Algol - au moyen de Lisp-Algol, naturellement. Il peut être utile à ceux qui s'intéressent à cette inclusion de Lisp dans Algol.

```
entier procédure EVALQUOTE (FN,X), valeur FN,X ; entier FN,X ;
       EVALQUOTE := APPLY (FN,X,1) ;

entier procédure APPLY (FN,X,A) ; valeur FN,X,A ; entier FN,X,A ;
       APPLY := si ATOM (FN) alors
                   (si FN=SCAR alors CAAR (X)
                               sinon
                    si FN=SCDR alors CDAR (X)
                               sinon
                    si FN=SCONS alors CONS (CAR (X), CADR (X))
                               sinon
                    si FN=SATOM alors ATOMM (CAR (X))
                               sinon
                    si FN=SEQ alors EQM (CAR (X), CADR (X))
                              sinon
                    APPLY (EVAL (FN,A), X, A)
                    )
                        sinon
                si CAR (FN) = SLAMBDA alors
                   EVAL (CADR (FN), PAIRLIS (CADR (FN), X, A))
                                    sinon
                si CAR (FN) = SLABEL alors
                   APPLY (CADDR (FN), X, CONS (LIST2 (CADR (FN), CADDR (FN)),A))
                                    sinon ERREUR ;

entier procédure LIST 2 (P,Q) ; valeur P,Q ; entier P,Q ;
       LIST 2 := CONS (P, CONS (Q,1)) ;
```

```
entier procédure EVAL (E,A) ; valeur E,A ; entier E,A ;
      EVAL := si ATOM (E) alors ASSOC (E,A)
                         sinon
              si ATOM (CAR (E)) alors
                 (si CAR (E) = SQUOTE alors CADR (E)
                                      sinon
                  si CAR (E) = SCOND alors EVCON (CDR (E), A)
                                     sinon
                  APPLY (CAR (E), EVLIS (CDR (E), A), A)
                  )
                         sinon
              APPLY (CAR (E), EVLIS (CDR (E), A), A) ;

entier procédure PAIRLIS (X,Y,A) ; valeur X,Y,A ; entier X,Y,A ;
      PAIRLIS := si X=1 alors A sinon
                   CONS (CONS (CAR (X), si ATOM (CAR (Y)) /\ CAR (Y) ≠1
                                        alors CAR (Y) sinon CONS(CAR(Y),1)),
                         PAIRLIS (CDR (X), CDR (Y), A)) ;

entier procédure ASSOC (X,A) ; valeur X,A ; entier X,A ;
      ASSOC := si EQUAL (CAAR (A), X) alors CADAR (A) sinon ASSOC (X,CDR(A)) ;

entier procédure EVCON (C,A) ; valeur X,A ; entier X,A ;
      EVCON := si EVAL (CAAR (C), A) ≠ 1 alors EVAL (CADAR (C), A)
               sinon EVCON (CDR (C), A) ;

entier procédure EVLIS (M,A) ; valeur M,A ; entier M,A ;
      EVLIS := si M=1 alors 1 sinon CONS (EVAL (CAR (M),A), EVLIS(CDR(M),A)) ;

entier procédure ATOMM (L) ; valeur L ; entier L ;
      ATOMM := si ATOM (L) alors TRUE sinon 1 ;
```

```
entier procédure EQM (A,B) ; valeur A,B ; entier A,B ;
      EQM := si A=B alors TRUE sinon 1 ;

entier TRUE,NIL,SCAR,SCDR,SCONS,SATOM,SEQ,SLAMBDA,SLABEL,SQUOTE,SCOND,
       PROGRAMME,DONNEE,STOP ;

       PROGRAMME PRINCIPAL :

       PREPARER (5000) ;

       TRUE    := NOMMER ('T') ;
       NIL     := NOMMER ('()') ;
       SCAR    := NOMMER ('CAR') ;
       SCDR    := NOMMER ('CDR') ;
       SCONS   := NOMMER ('CONS') ;
       SATOM   := NOMMER ('ATOM') ;
       SEQ     := NOMMER ('EQ') ;
       SLAMBDA := NOMMER ('LAMBDA') ;
       SLABEL  := NOMMER ('LABEL') ;
       SCOND   := NOMMER ('COND') ;
       STOP    := NOMMER ('STOP') ;

   boucle : PROGRAMME := LIRE LISTE ;
            si PROGRAMME ≠ STOP alors
               début
                   DONNEE := LIRE LISTE ;
                   ECRIRE LISTE (EVALQUOTE (PROGRAMME, DONNEE)) ; allera boucle
               fin
   fin
```

V.5.3. Transformation en arbre du résultat d'une analyse

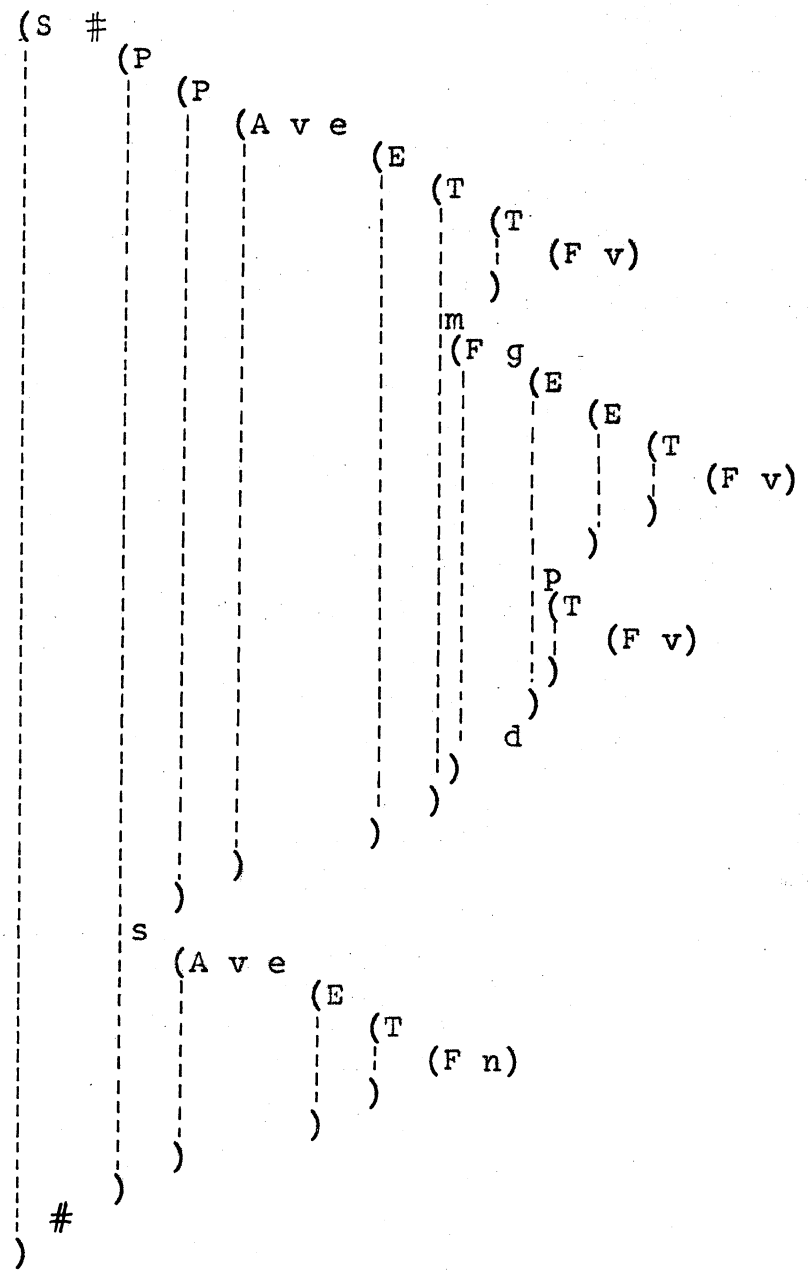
Nous présentons ici la procédure CONSTRUCTION, qui permet de transformer en un arbre, au sens employé dans ce chapitre, le résultat de l'analyse syntaxique ascendante décrite en § IV.2.4. Les données utilisées par CONSTRUCTION sont :

a. la pile TROUVE et son pointeur "niveau".

b. les règles de la grammaire représentées par les tableaux TERMINAL, ARBRE et PRECEDANT (cf. § IV.2.4).

Remarquons que cette procédure est fonctionnellement analogue à la procédure ARBRE du § V.5.1. Une caractéristique importante de CONSTRUCTION est qu'elle construit le plus grand arbre disponible au sommet de TROUVE et qu'elle laisse dans la partie inférieure de cette pile les métavariables pour lesquelles on n'a pas encore trouvé toutes les branches nécessaires à la construction d'un arbre ou sous-arbre. Cette caractéristique est indispensable si l'on veut pouvoir alterner analyse et génération au moyen des points de génération (cf. § IV.5 et VI.2).

La procédure CONSTRUCTION appliquée à l'exemple donné aux § IV.2.3 et § IV.2.4 produit comme résultat l'arbre suivant :


(S #
(P
(P
(A v e
(E
(T
(T
(F v)
)
m
(F g
(E
(E
(T
(F v)
)
)
P
(T
(F v)
)
)
d
)
)
)
)
)
s
(A v e
(E
(T
(F n)
)
)
)
)
#
)

```
entier procédure CONSTRUCTION ;
    début entier M,t,q ;
        t := TROUVE [niveau + 4] ;
        M := ARBRE [niveau] ;
        q := 1 ;
    E1 : t := PRECEDANT [t] ;
        si t ≠ DEUX ETOILES alors
            début
                si TERMINAL [ARBRE [t]]
                  alors q := CONS (ARBRE [t], q)
                  sinon début
                            niveau := niveau - 5 ;
                            q := CONS (CONSTRUCTION, q)
                        fin ;
                  allera E1
            fin ;
      CONSTRUCTION := CONS (M,q)
    fin CONSTRUCTION ;
```

V.5.4. Transformation d'un arbre d'analyse en un arbre de dépendance

La procédure CONDENSATION* transforme un arbre d'analyse syntaxique en un arbre de dépendance selon l'algorithme décrit au § IV.4. Ses paramètres sont l'arbre d'analyse L généré par exemple au § V.5.3, et la liste d'atomes numériques R qui indique les inversions à effectuer. Rappelons qu'un atome numérique entre 1 et 9 est représenté par un code entre 31 et 39 (cf. procédure LIRE SYMBOLE au § V.1) ; RTABLE [m] contient la liste d'atomes numériques correspondant à la métavariable dont le code est m ; tp est une variable temporaire globale.

Pour la grammaire :

		1	2	3	
<inst. aff.>	::=	<variable>	:=	<exp. arith.>	(2 1 3)
<exp. arith.>	::=	<terme>			(1)
<exp. arith.>	::=	<exp. arith.>	+	<terme>	(2 1 3)
<terme>	::=	<facteur>			(1)
<terme>	::=	<terme>	*	<facteur>	(2 1 3)
<facteur>	::=	<variable>			(1)
<facteur>	::=	(	<exp. arith.>	)	(2)
<variable>	::=	A | B | C | D			

L'analyse de la chaîne A := B + C * D produit comme résultat l'arbre L suivant :

(*) Signalons qu'on ne peut obtenir d'arbres ayant plus de branches que le nombre exprimant la longueur des règles, moins un. Cette restriction pourrait facilement être levée au moyen d'un symbole spécial, placé devant l'entier qui correspond à un noeud de l'arbre de dépendance.

```
(inst. aff  (variable A)
            :=
            (exp. arith  (exp. arith  (terme  (facteur  (variable B))))
                         +
                         (terme  (terme  (facteur  (variable C)))
                                 *
                                 (facteur  (variable D))
                         )
            )
)
```

Un appel de CONDENSATION avec comme premier paramètre cet arbre L et comme deuxième paramètre les listes numériques indiquées à droite de la définition sous FNB produit l'arbre suivant :

(:= A (+ (B (* C D))))

```
entier procédure CONDENSATION (L) règle : (R) ; valeur L,R ; entier L,R ;
    début
        si R ≠ 1 alors
          début
            tp := TCAR [TCDR [CAR (R)]] ;
            si tp > 30 ∧ tp < 40 alors
               début commentaire atome numérique : réordonnancement ;
                 tp := IEME (CDR (L), tp-30) ;
                 si ¬ ATOM (tp) alors
                    début commentaire condenser la liste correspondante
                                      à cette métavariable ;
                      tp := CONDENSATION (tp,RTABLE [TCAR [TCDR [CAR (tp)]]]) ;
                      commentaire si la liste condensée obtenue est
                                  réduite à un seul élément prendre
                                  directement cet élément ;
                      si CDR (tp) = 1 alors tp := CAR (tp)
                    fin ;
                 CONDENSATION := CONS (tp, CONDENSATION (L, CDR (R)))
               fin sinon atome non numérique à insérer :
            CONDENSATION := CONS (CAR (R), CONDENSATION (L, CDR (R)))
          fin sinon
          CONDENSATION := 1
    fin procédure CONDENSATION ;

entier procédure IEME (L,I) ; valeur L,I ; entier L,I ;
    IEME := si I=1 alors CAR (L) sinon IEME (CDR (L), I-1) ;
```

VI. Génération

"Les mots diversement rangés font un divers sens. Et les sens diversement rangés font différents effets."

(Pascal, Pensées)

"Occupez-vous du sens, les mots se débrouilleront bien tout seuls."

(Lewis Carroll, Alice au pays des merveilles)

Nous nous occupons dans ce chapitre des moyens permettant la transformation du résultat de l'analyse syntaxique en une chaîne de symboles pouvant être exécutée (ou interprétée) en machine. Rappelons que le but de l'analyse syntaxique a été de transformer une chaîne linéaire en une chaîne parenthésée, arbre syntaxique ou arbre de dépendance. En revanche, le but de la génération est d'effectuer une opération de type inverse : transformer la chaîne parenthésée en une chaîne linéaire représentant le programme objet. Afin d'accomplir cette dernière transformation au moyen d'un langage de génération, on doit donner à ce dernier les caractéristiques suivantes :

1. Il doit inclure des moyens de reconnaissance des structures imbriquées que l'on trouve dans les arbres de dépendance ; la récursivité sera donc appelée à jouer de nouveau un rôle important.

2. Compte tenu de l'importance attribuée aux opérateurs, il est intéressant de pouvoir s'aiguiller directement vers une partie du programme de génération relative à l'opérateur occupant le noeud de l'arbre (ou sous-arbre) que l'on veut traiter.

3. Il faut créer un nouveau type de quantité pour représenter une suite d'instructions du programme objet. Dans le langage décrit au § VI.2 nous appelons ce nouveau type "segment". Le langage doit aussi permettre la concaténation des segments et l'insertion ultérieure des informations qui n'ont pas pu être obtenues lors de la génération d'un segment donné.

Ces trois caractéristiques font partie du langage de génération proposé au § VI.2. Nous présentons en Lisp-Algol au § VI.1 un exemple d'évaluation des expressions arithmétiques, qui sert à justifier l'inclusion de ces caractéristiques dans le langage de génération. On trouvera en [Bol] un autre exemple, lui aussi en Lisp-Algol, de génération à partir de l'arbre de dépendance ; dans ce dernier cas le programme généré est formé de "macros-instructions".

Au paragraphe VI.3, nous présentons une machine abstraite, qui traduit automatiquement un arbre de dépendance en un programme objet écrit dans le langage d'une machine dont les instructions sont

décrites au moyen de règles de réécriture. Bien qu'incomplet, ce dernier travail ouvre une voie nouvelle vers la paramétrisation des compilateurs, le côté original étant la description de la machine pour laquelle on veut générer le programme objet.

Il est juste de signaler que ce chapitre ne présente que de façon succincte le langage de génération et les raisons qui nous ont conduit à le définir. Cela a été surtout fait pour donner une vue d'ensemble du système que nous proposons. Une présentation complète, contenant des exemples, sera faite par Laurent TRILLING en [Tr].

Nous regrettons que la littérature portant sur la génération soit si peu abondante, surtout en comparaison avec le grand nombre des publications relatives à l'analyse syntaxique. Parmi les plus intéressants des travaux sur la génération, signalons [Je, Ge, Gra]. En particulier, le langage proposé par GRAHAM [Gra] nous a influencé pendant l'élaboration du langage de génération.

VI.1. Exemple d'utilisation du résultat de l'analyse

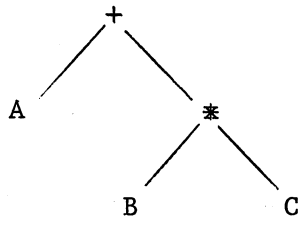
Nous justifions ici au moyen d'un exemple les caractéristiques principales du langage de génération décrit au § VI.2. Le programme en Lisp-Algol présenté plus loin permet l'évaluation des expressions arithmétiques données sous forme d'arbres de dépendance (obtenus, par exemple, comme résultat du programme décrit au § V.5.4). Une étude attentive de cet exemple aidera à comprendre la raison d'être du mécanisme adopté pour la génération.

A première vue, le programme paraîtra peut-être inutilement compliqué ; cela est dû à la symétrie que nous avons volontairement introduite pour pouvoir traiter d'autres cas plus complexes et faciliter la compréhension du paragraphe suivant.

La procédure EVALUATION fournit la valeur d'une expression arithmétique donnée comme un arbre de dépendance formé des opérateurs +, -, ×, / et des opérandes A, B, C, D ... ; les valeurs de ces derniers ont été préalablement rangées dans le tableau VALEUR ; i étant le code d'un opérande sa valeur est VALEUR [i] (cf. § V.1). Aux opérateurs +, -, ×, / on fait correspondre les codes 1, 2, 3 et 4 au moyen des valeurs préalablement rangées dans le tableau TA. La procédure EVALUATION utilise deux procédures auxiliaires EVAL et EVAL LISTE. La première calcule la valeur d'un atome, celui de gauche ou celui de droite selon que n est égal à 1 ou 2. Remarquons que les opérateurs ont été considérés comme binaires, mais que dans le cas

général ils pourront être n-aires. EVAL LISTE appelle EVALUATION pour calculer la valeur du sous-arbre placé à gauche ou à droite d'un opérateur donné.

Le programme commence par s'aiguiller vers l'étiquette associée à l'opérateur que l'on doit traiter. La procédure "profil" permet ensuite d'aller vers l'une des étiquettes e 00, e 01, e 10 ou e 11. La suite de 0 et de 1, considérée de gauche à droite, est une représentation respectivement de l'atomicité ou de la non-atomicité des opérandes considérés eux aussi de gauche à droite. Ainsi l'expression arithmétique représentée par



est traitée par la séquence d'instructions suivant l'étiquette e 01. On voit que la discrimination selon ce critère s'étend aussi aux opérateurs n-aires.

```
reel procedure EVALUATION (L) ; valeur L ; entier L ;
    debut aiguillage opération := op 1, op 2, op 3, op 4 ;
        reel procedure EVAL (A,n) ; valeur A,n ; entier A,n ;
            debut aiguillage B := b1, b2 ;
                allera B [n] ;
                b1 : EVAL := VALEUR [TCAR [TCDR [CADR (A)]]]; allera fin ;
                b2 : EVAL := VALEUR [TCAR [TCDR [CADDR (A)]]];
                fin :
            fin EVAL ;
        reel procedure EVAL LISTE (L,n) ; valeur L,n ; entier L,n ;
            debut aiguillage A := a1, a2 ;
                allera A [n] ;
                a1 : EVAL LISTE := EVALUATION (CADR (L)) ; allera fin ;
                a2 : EVAL LISTE := EVALUATION (CADDR (L)) ;
                fin :
            fin EVAL LISTE ;
          allera opération [TA [TCAR [TCDR [CAR (L)]]]];

    op 1 : debut aiguillage E := e 00, e 01, e 10, e 11 ;
              allera E [profil (CDR (L))] ;
              e 00 : EVALUATION := EVAL (L,1) + EVAL (L,2) ; allera fin ;
              e 01 : EVALUATION := EVAL (L,1) + EVAL LISTE (L,2) ; allera fin ;
              e 10 : EVALUATION := EVAL LISTE (L,1) + EVAL (L,2) ; allera fin ;
              e 11 : EVALUATION := EVAL LISTE (L,1) + EVAL LISTE (L,2) ;
              fin : allera finale
           fin op 1 ;
```

```
        op 2 : debut aiguillage E := e 00, e 01, e 10, e 11 ;
                   allera E [profil (CDR (L))] ;
                   e 00 : EVALUATION := EVAL (L,1) - EVAL (L,2) ; allera fin ;
                   e 01 : EVALUATION := EVAL (L,1) - EVAL LISTE (L,2) ; allera fin ;
                   e 10 : EVALUATION := EVAL LISTE (L,1) - EVAL (L,2) ; allera fin ;
                   e 11 : EVALUATION := EVAL LISTE (L,1) - EVAL LISTE (L,2) ;
                   fin : allera finale ;
               fin op 2 ;

        op 3 : debut commentaire le bloc correspondant à op 3 est analogue à
                                 op 1 où 1'on remplace le symbole "+" par
                                 "x" ;
                  ...
               fin op 3 ;

        op 4 : debut commentaire le bloc correspondant à op 4 est analogue à
                                 op 1 où 1'on remplace le symbole "+" par
                                 "/" ;
                  ...
               fin op 4 ;

        finale :

    fin EVALUATION ;


entier procedure profil (L) ; valeur L ; entier L ;
    debut entier s ;
        s := 0 ;
     E : s := si ATOM (CAR (L)) alors s × 2 sinon s × 2 + 1 ;
        si CDR (L) ≠ 1 alors debut
                                L := CDR (L) ;
                                allera E
                            fin ;
        profil := s + 1
    fin profil ;
```

VI.2. Langage de génération

VI.2.1. Programme

Syntaxe :

<programme génération> ::= <déclaration> & <corps> | corps

Sémantique :

Un programme> de génération opère sur une donnée dont la structure est une structure d'arbre (cf. § V.4). Il fournit en résultat la valeur des segments déclarés dans le programme (cf. § VI.2.8 et § VI.2.2).

VI.2.2. Déclaration

Syntaxe :

<déclaration> ::= <déclaration unique> | <déclaration unique>, <déclaration>
déclaration unique> ::= <type> <identificateur> | local <type> <identificateur>
type> ::= entier | logique | segment

Sémantique :

Les variables, représentées par les identificateurs, et de type entier ou logique prennent respectivement leurs valeurs sur les nombres entiers ou les valeurs logiques.

La valeur d'une variable de type segment est une suite, pouvant être vide, d' instructions objet (cf. § VI.2.6). Cette suite est formée à l'aide des actions générer et remplir (cf. § VI.2.5). La notation adoptée

pour la suite attachée au segment p, à un instant donné de l'exécution, est la suivante :

$$u_1^p \quad u_2^p \quad \ldots \quad u_i^p \quad \ldots \quad u_m^p$$

où u représente une <instruction objet> ; m est "l'indice maximal" de la suite. Si la variable déclarée est spécifiée "local", sa valeur est déterminée par l'emploi des actions ancien et nouveau (cf. § VI.2.5).

Exemple :

entier A, local segment B, segment C

VI.2.3. Corps

Syntaxe :

<corps> ::= <chapitre># | <chapitre> # <corps>

<chapitre> ::= e <opérateur> : <corps de chapitre> |
* e <opérateur> : <corps de chapitre>

<corps de chapitre> ::= <sous-chapitre> | <sous-chapitre> ; <corps de chapitre>

<sous-chapitre> ::= <tête> : <suite d'instructions>

<tête> ::= e <profil> | e

<opérateur> ::= <symbole> | <symbole> <opérateur>

<profil> ::= <01> | <01> <profil>

<01> ::= 0 | 1

<suite d'instructions> ::= <instruction> | <instruction> ;
<suite d'instructions>

Sémantique :

Le corps de chaque <chapitre> représente le travail à effectuer (éventuellement) si l'arbre "courant" possède une racine dénotée par l' opérateur associé au <chapitre>. Nous désignons par arbre "courant", soit l'arbre fourni en donnée*, soit l'arbre fourni par une action "traiter" (cf. § VI.2.5). Dans ce dernier cas, l'exécution du <chapitre> est toujours faite ; dans le premier cas, elle ne l'est que si le <chapitre> débute par le symbole "*".

Un <chapitre> est divisé en <sous-chapitres>, chacun d'eux débutant par une <tête>. Seule la <suite d'instructions> correspondant à un de ces <sous-chapitres> est exécutée lorsqu'on atteint le noeud considéré. Ces instructions s'exécutent séquentiellement. Le <sous-chapitre> en question est :

- soit celui dont la <tête> contient un profil "acceptable".
- soit celui dont la <tête> ne contient pas de profil.

Ce <sous-chapitre> doit être unique. Le profil est dit "acceptable" s'il est identique à la suite de 0 et de 1 obtenue de la façon suivante : considérant successivement le "fils" du noeud traité et ses "frères" (cf. § VI.2.7), on construit de gauche à droite une suite de 0 et de 1, en créant un 0 si l'élément considéré n'a pas de descendant et un 1 dans le cas contraire.

(*) C'est-à-dire l'arbre de dépendance donné par le programme d'analyse lorsque le contrôle passe de ce programme au programme de génération (cf. rôle de G au § IV.5).

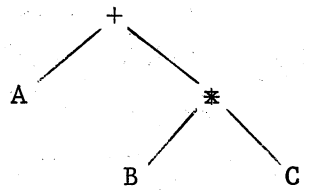
Exemple :

<chapitre> :

* e + :

e00 : <suite d'instructions> ;
e01 : < " > ;
e10 : < " > ;
e11 : < " > ;

arbre :



Le sous-chapitre pris en considération est celui dont la tête est e01.

VI.2.4. Instruction

Syntaxe :

<instruction> ::= <étiquette> : <instruction de base> | <instruction de base>

<instruction de base> ::= <instruction d'affectation> |
<instruction allera> |
<instruction conditionnelle> |
<action>

<instruction d'affectation> ::= <variable> := <expression>

<expression> ::= <expression entière> | <expression logique>

<instruction allera> ::= allera <étiquette>

<instruction conditionnelle> ::= si <expression logique> alors
<instruction allera>

Sémantique

Les instructions d'affectation, aller à et conditionnelle, ont la même semantique qu'en Algol 60 (avec les restrictions apportées par la syntaxe).

VI.2.5. Actions

Syntaxe :

<action> ::= générer (, <instruction objet>) |
générer () |
traiter (<descendant>) |
garder (,) | annuler ()
remplir (,)|
nouveau (<variable locale>)|
ancien (<variable locale>) |
établir description (<descendant>, <suite d'expressions entières>)

<suite d'expressions entières> ::= <expression entière> |
<expression entière>,
<suite d'expressions entières>

<variable locale> ::= <variable entière locale> |
<variable logique locale> |
<segment local>

Sémantique :

L'action traiter permet de suspendre momentanément l'exécution du chapitre courant, et de se référer au chapitre correspondant au noeud désigné par le descendant (cf. § VI.2.7) associé à l'action traiter. On revient ensuite exécuter le reste du chapitre primitif.
L'action générer a pour but d'ajouter l'<instruction objet> indiquée à la suite représentée par le segment associé à cette action. Si la suite est dénotée (cf. § VI.2.2) :

$$u_1^p \quad u_2^p \quad \ldots \quad u_m^p \quad ,$$

on ajoute l' instruction objet u_{m+1}^p pour que la suite devienne :

$$u_1^p \quad u_2^p \quad \ldots \quad u_m^p \quad u_{m+1}^p \quad .$$

remplir est une action destinée à intercaler, dans la suite désignée par le premier segment cité, la suite désignée par le second. L'indice où débute ce placement est l'indice maximal de la suite, correspondant au second segment à l'exécution de l'action garder précédente qui comportait les mêmes segments associés. Il faut donc, pour pouvoir utiliser remplir, avoir auparavant activé au moins une action garder composée des mêmes segments. Soit p0 le premier segment, p1 le second et i l'indice maximal dont il est question. Si

$$u_1^{p0} \ \ldots \ u_i^{p0} \ u_{i+1}^{p0} \ \ldots \ u_m^{p0}$$

et

$$u_1^{p1} \ \ldots \ u_k^{p1}$$

sont respectivement les dénotations des suites p0 et p1 avant l'action remplir les concernant, l'effet de celle-ci est de transformer la suite p0 en

$$v_1^{p0} \ \ldots \ v_i^{p0} \ v_{i+1}^{p0} \ \ldots \ v_{i+k}^{p0} \ v_{i+k+1}^{p0} \ \ldots \ v_{m+k}^{p0}$$

dans laquelle v_j^{p0} est une instruction objet

$$v_j^{p0} = u_j^{p0} \quad \text{pour} \quad 1 \leqslant j < i$$

$$v_{i+j}^{p0} = u_j^{p1} \quad \text{pour} \quad 1 \leqslant j \leqslant k$$

$$v_{i+k+j}^{p0} = u_{i+j}^{p0} \quad \text{pour} \quad 1 \leqslant j \leqslant m$$

annuler rend vide la suite représentée par le segment associé.
L'action nouveau indique que l'identificateur représentant la variable déclarée locale correspondante représente désormais une nouvelle variable.

L'identificateur désignera à nouveau la première variable à la prochaine exécution de l'action ancien s-y rapportant.

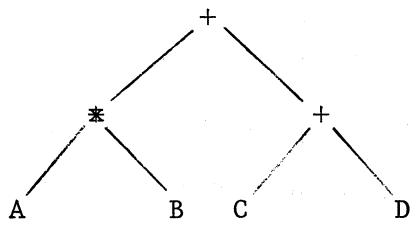
établir description crée un bloc d'informations dont le nom est celui du noeud désigné par le <descendant> indiqué (cf. § VI.2.7). Ces informations sont données par la <suite d'expressions entières> ; elles pourront être atteintes ultérieurement à l'aide de la valeur entière descripteur (cf. § VI.2.10).

Exemples :

a. On peut écrire le chapitre :

```
e+ :
  e11 : traiter (* i *) ;
        traiter (* ri *) ;
        générer (segment 1, 'add')
```

L'action de ce chapitre quand on traite l'arbre :



conduit à considérer les deux sous-arbres dépendant du noeud "+", et à ajouter dans la suite désignée par "segment 1" l'instruction objet "add".

b. Soient α et β deux opérateurs. Supposons qu'un noeud désigné par soit traité avant celui désigné par β. On désire générer pendant

tout le traitement sur le segment p0 puis, à l'apparition de β, générer sur le segment p1 et intercaler le contenu de p1 dans p0 à l'endroit correspondant à la première génération du chapitre de α. On peut écrire :

```
e α :
    e <    > : garder (p0, p1) ;                          (1)
               générer (p0, '<          >') ;
               traiter (*<     >*)#
e β :
    e <    > : générer (p1, '<          >') ;
               remplir (p0, p1) ;                         (2)
               traiter (*<     >*)#
```

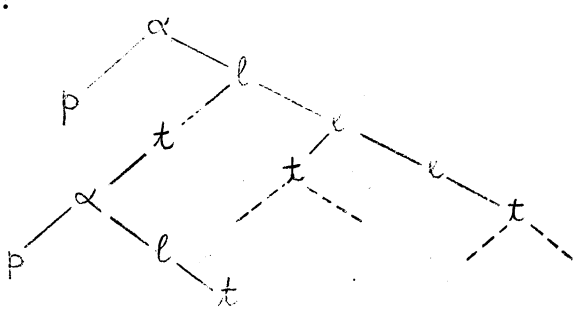
En conservant les mêmes notations qu'au § VI.2.5, on note que :

i est l'indice maximal de p0 en (1)

k est l'indice maximal de p1 en (2)

c. Soit à traiter un arbre de racine représentée par l'opérateur α et possédant des sous-arbres ayant une racine identique. Si on suppose que le traitement de α nécessite par exemple l'emploi d'une variable entière, celle-ci doit être nouvelle à chaque traitement de α. Elle sera alors déclarée locale.

Considérons l'arbre :



Si l'on désire que la variable entière et locale k prenne dans le chapitre correspondant à α une valeur égale au nombre de t non contenus dans un sous-arbre de sommet α , on peut écrire :

```
e∝ :
    e01 : nouveau (k) ;
          traiter (* ri *) ;
          ...
          ancien (k) #
et :
    e :   traiter (* i *) ;
          k := k+1 #
```

VI.2.6. Instruction objet

Syntaxe :

<instruction objet> ::= <suite d'éléments>

<suite d'éléments> ::= <élément> | <suite d'éléments> <élément>

<élément> ::= <élément 1> | . <élément 1>

<élément 1> ::= eg <expression entière> | '<chaine>' |
<expression entière>

Sémantique :

Les <instructions objet> peuvent être formées de la concaténation de chaînes qui représentent des éléments du langage objet et des valeurs d'expressions entières calculées dans le programme de génération.

Un "." figurant en tête d'un élément signifie qu'on augmente de 1 le compteur (cf. § VI.2.10) associé au segment considéré dans la création de l'⟨instruction objet⟩. eg précise que l'élément formé est une étiquette implicite du programme généré (cf. § VI.2.10).

Exemples :

'allera E' x

'si t=' n+3 'alors t' x ':=1'

VI.2.7. Descendant

Syntaxe :

⟨descendant⟩ ::= * ⟨suite ir⟩ * | noeud
⟨suite ir⟩ ::= ⟨composition ir⟩ | ⟨composition ir⟩ ⟨suite ir⟩
⟨composition ir⟩ ::= ⟨fils frère⟩ | ⟨fils frère⟩ ⟨expression entière⟩
⟨fils frère⟩ ::= i | r

Sémantique :

Un ⟨descendant⟩ représente un noeud dans l'arbre déterminé par le noeud considéré dans un chapitre. Un fils (plus exactement un fils ainé) est une fonction fournissant le noeud immédiatement dérivé de la première branche du noeud considéré (cf. § V.4). De même, un frère (plus exactement un frère puîné) fournit, si le noeud considéré est immédiatement dérivé de la i-ième branche, le noeud dérivé de la i+1-ième branche.

Une <composition ir> est la composition de n fils ou de n frères (n étant entier). Une <suite ir> est une composition des précédentes fonctions. Noeud désigne le noeud considéré. On note que l'expression entière attachée à une <composition ir> doit toujours être positive.

Exemple :

* r 2 i *

désigne le frère du frère du fils du noeud considéré.

VI.2.8. Segment

Syntaxe :

 ::= <segment global> | <segment local>
<segment local> ::= <identificateur> | <identificateur> moins <expression entière>
<segment global> ::= <identificateur>

Sémantique :

Un segment local est un segment déclaré local. On a vu au § VI.2.5 qu'il est possible à l'aide de l'action nouveau de créer dynamiquement un nouveau segment.

A l'aide de l'expression entière (toujours positive) attachée à un segment local, on pourra atteindre un segment créé auparavant. La valeur n de l'expression entière permet d'atteindre le segment depuis lequel on en a créé n autres.

Exemple :

Soient les déclarations :

```
segment p0 ;
local segment p1 ;
```

et les chapitres :

```
e α : ...
      nouveau (p1) ;
      garder (p0, p1) ;
      traiter (*<   >*) ;
      ancien (p1) #
e β : ...
      remplir (p0, p1) ;
      remplir (p0, p1 moins 1) #
```

Si on examine successivement deux noeuds α puis un noeud β, le contenu des deux derniers segment p1 créés dans le chapitre α est intercalé dans p0 au cours du traitement associé au noeud β.

VI.2.9. Expressions logiques et entières

Syntaxe :

```
<variable entière globale> ::= <identificateur>
<variable entière locale>  ::= <identificateur>
<variable logique globale> ::= <identificateur>
<variable logique locale>  ::= <identificateur>
```

Les <expressions logiques> et les <expressions entières sont respecti-

vement les expressions booléennes simples et les expressions arithmétiques simples d'Algol 60 avec les modifications suivantes :

<primaire arithmétique> ::= <valeur entière> |
<variable entière> |
(<expression entière>)

<variable entière> ::= <variable entière globale> | <variable entière locale>

<primaire booléen> ::= <valeur logique> |
<variable logique> |
<relation> |
(<expression logique>)

<variable logique> ::= <variable logique globale> | <variable logique locale>

Sémantique :

Les opérateurs arithmétiques et booléens ont les mêmes significations qu'en Algol 60. Les relations sont aussi composées de la même façon.

VI.2.10. Valeurs entières et logiques

Syntaxe :

<valeurs entières> ::= nombre |
descripteur (<descendant>, <expression entière>) |
compteur () |
<nombre entier>

<valeurs logiques> ::= vrai | faux |
terminal (<descendant)

Sémantique :

nombre, mis à jour dans chaque chapitre, fournit le nombre de branches du noeud considéré.

compteur donne la valeur courante du compteur ordinal des ⟨instructions objets⟩ créées sur un segment donné. On augmente la valeur de ce compteur en plaçant un point dans une instruction objet (cf. § VI.2.6).

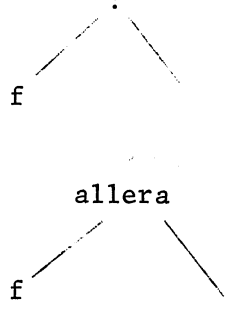
La valeur n de l' expression entière toujours positive associée à un descripteur permet d'atteindre la valeur du n-ième composant de la description dont le descendant est identique (cf. établir description, § VI.2.5).

terminal prend la valeur faux si le descendant considéré n'a lui-même aucun descendant, vrai dans le cas contraire.

Exemple :

Supposons que "allera" et ":" soient des opérateurs possédant des étiquettes comme opérandes. On désire traduire une instruction allera ALGOL 60 dans un langage objet ne possédant pas d'adressage relatif. On peut écrire :

```
segment p0 ;
...
e ::e01 : établir description (* i *, compteur (p0)) ;
          traiter (* ri *)
...
e allera : e0 : générer (p0, 'tra' eg descripteur
                                    (* i *, 1))
```



VI.2.11. Nombres entiers, étiquettes, symboles, identificateurs

Les nombres entiers et les identificateurs sont définis comme en Algol 60. Les étiquettes sont représentées comme des identificateurs. Quant aux symboles, ils peuvent représenter tous les éléments de l'alphabet dont on dispose.

VI.3. Description d'un transducteur traduisant en langage machine une chaîne donnée sous forme préfixée

Le seul langage que nous connaissons qui soit spécialisé dans la description des machines est celui d'IVERSON [Iv]. Mais son objectif est de préciser dans le détail l'action d'une instruction, alors que nous nous intéressons aux successions possibles d'ordres-machine nécessaires pour aboutir à un résultat donné. En effet, notre problème est de rechercher les séquences d'instructions représentant la traduction d'une chaîne en langage source.

Nous ne dissimulerons pas l'extrême complexité du problème. Notre but est essentiellement de présenter une voie nouvelle qui, nous le pensons, permet d'envisager cette question sous un autre aspect, plus précisément d'un point de vue combinatoire.

A cet effet, nous définirons dans ce qui suit un transducteur auquel on fournit :

- une chaîne d'entrée dans laquelle les opérateurs sont distingués et placés devant les opérandes associés, résultat aisément obtenu par l'utilisation du langage d'analyse (§ IV.5) ;
- une description des instructions du calculateur pour lequel on désire un programme objet.

Le transducteur produit une séquence d'instructions représentant la traduction de la chaîne d'entrée. A cet effet, on doit connaître les opérateurs dont on dispose dans le répertoire des instructions du calculateur. En général, certaines de celles-ci correspondent

à des opérateurs se trouvant dans la chaîne d'entrée. Si un opérateur choisi par l'utilisateur ne se trouve associé à aucune instruction, on doit "s'efforcer" de lui faire correspondre une séquence d'opérateurs de la machine : l'introduction de règles spéciales permet d'effectuer cette transformation.

La chaîne d'entrée de notre transducteur a la syntaxe suivante :

⟨chaîne⟩ ::= ⟨opérateur⟩ ⟨suite d'opérandes⟩ | ⟨atome⟩
⟨suite d'opérandes⟩ ::= ⟨chaîne⟩ | ⟨chaîne⟩ ⟨suite d'opérandes⟩

Un ⟨opérateur⟩ représente les opérateurs mis en évidence lors de l'analyse syntaxique ; on doit les retrouver associés au même nombre d'opérandes dans les règles présentées plus loin. Un ⟨atome⟩ désigne un opérande élémentaire ou un identificateur au sens d'Algol.
Par exemple :

```
:= A + * B C / D E        et
+ A sin PHI
```

sont des chaînes données acceptables par notre transducteur.

Le répertoire des instructions de la machine pour laquelle on désire générer le programme objet se construit au moyen de règles de réécriture du transducteur. La syntaxe de ces règles est la suivante :

⟨règle⟩ ::= ⟨partie gauche⟩ ⩾ ⟨partie droite⟩
⟨partie droite⟩ ::= λ | ⟨suite d'expressions⟩
⟨partie gauche⟩ ::= ⟨expression⟩
⟨suite d'expressions⟩ ::= ⟨expression⟩ | ⟨expression⟩ et ⟨suite d'expressions⟩
⟨expression ::= ⟨registre⟩ = ⟨terme⟩|⟨terme⟩
⟨terme⟩ ::= ⟨opérateur⟩ ⟨suite d'opérandes 1⟩

<suite d'opérandes 1> ::= <opérande 1> | <suite d'opérandes 1> <opérande 1>

<opérande 1> ::= T | M | ω_i

où

ω_i	désigne une chaîne du type défini par <chaîne>
M	désigne un atome
T	désigne une mémoire de travail
<registre>	désigne les registres de la machine sur lesquels opèrent les instructions
$\gg$	veut dire "se réécrit en"
λ	représente l'expression vide
et	est un opérateur de concaténation pour les expressions
=	est un symbole permettant une meilleure lecture du nom du registre associé à un terme.

A chaque instruction correspond une règle de ce type. Par exemple, à l'instruction ADD M on attache la règle

$$Acc = +M\ \omega_o \gg Acc = \omega_o$$

que l'on interprète ainsi : "Pour que l'instruction soit utilisée, il faut que l'un des opérandes (ω_o) soit rangé dans le registre Acc ; le but de l'opération est d'additionner M à ω_o".

Passons maintenant au fonctionnement du transducteur. Définissons tout d'abord l'applicabilité d'une partie gauche β à une séquence de symboles . β est applicable à α si à chaque élément de β on peut faire correspondre dans le même ordre de succession une partie de α,

- soit identique à l'élément de β ,
- soit désignée par l'élément de β .

Exemple : la partie gauche Acc = + M ω_o est applicable à Acc = + A * BC.
Etant donnée une expression fournie au transducteur, celui-ci recherche parmi les règles de réécriture celle(s) dont la partie gauche est applicable à cette expression. Dans un cas favorable, une double tâche est dévolue au transducteur :

a) Transformation de l'expression donnée

La nouvelle expression considérée est obtenue en faisant prendre aux symboles ω_i, M et T de la partie droite de la règle utilisée leurs valeurs trouvées lors du calcul d'applicabilité.

Exemple : L'emploi de la règle Acc = + M ω_o $\succ$ Acc = ω_o transforme l'expression Acc = + A * BC en Acc = * BC avec ω_o désignant * BC et M désignant A.

Si l'expression est transformée en une suite d'expressions concaténées par le symbole et, le transducteur opère de façon analogue sur chacune de ces expressions prises de gauche à droite.

b) Génération d'une instruction

Si une instruction est associée à la règle utilisée, les valeurs de ses constituants sont celles qui sont prises dans le calcul d'applicabilité par les élément précisés comme constituants de l'instruction.

Exemple : l'instruction ADD M attachée à la règle de l'exemple précédent aboutit à la génération de l'instruction ADD A.

La première expression fournie au transducteur est la chaîne donnée. Elle est progressivement réduite par applications des règles de réécriture. La génération du programme traduisant cette chaîne (ou programme objet) est terminée lorsque la chaîne est réduite au vide. Les instructions de ce programme sont évidemment générées dans l'ordre inverse de celui de leur exécution.

Il est possible que plusieurs règles puissent être appliquées à la réduction d'une même expression. Cela veut dire que plusieurs programmes objets peuvent être la traduction d'une même chaîne donnée, c'est-à-dire que le transducteur n'est pas déterministe.

VI.3.1. Exemple d'utilisation

On suppose le répertoire des instructions de la machine défini par les règles suivantes :

n°	instruction	règle du transducteur
1	CLA M	Acc = M $\gg \lambda$
2	LDQ M	Mq = M $\gg \lambda$
3	ADD M	Acc = + ω_o M $\gg$ Acc = ω_o
4	ADD M	Acc = + M ω_o $\gg$ Acc = ω_o
5	SUB M	Acc = - ω_o M $\gg$ Acc = ω_o
6	MTP M	Acc = * ω_o M $\gg$ Mq = ω_o
7	MTP M	Acc = * M ω_o $\gg$ Mq = ω_o
8	DIV M	Mq = / $_o$ M Acc = ω_o
9	STO M	:= M ω_o $\gg$ Acc = ω_o
10	STQ M	:= M ω_o $\gg$ Mq = ω_o

Soit la chaîne d'entrée := F/+A*BCD. Les successions de règles aboutissant à la réduction au vide de cette chaîne sont : 10,8,4,7,2 et 10,8,4,6,2 qui produisent respectivement les programmes objets :

```
LDQ B          et      LDQ C
MTP C                  MTP B
ADD A                  ADD A
DIV D                  DIV D
STQ F                  STQ F
```

On note que le transducteur a essayé d'employer pour réduire la chaîne initiale la règle 9 : celle-ci a produit la nouvelle expression "Acc = / + A * BCD" à laquelle aucune partie gauche de règle n'est applicable. Le transducteur n'a pu poursuivre dans cette voie et est revenu en arrière pour appliquer la règle 10.

VI.3.2. Problèmes posés par l'extension du transducteur

Le transducteur précédemment présenté ne permet de traiter que des expressions relativement simples. Plusieurs moyens d'extension sont possibles :

1. Utilisation de mémoires de travail

   On peut les introduire à l'aide de règles analogues aux règles décrites antérieurement : elles transforment une expression non réductible par les règles décrivant les instructions, en une expression équivalente réductible, employant des mémoires

de travail. Aucune instruction ne correspond à ces règles.

Par exemple, les deux règles de ce genre utilisées pour la machine précédente sont les suivantes :

a) $R = op\ \omega_1\ \omega_2 \gg R = op\ \omega_1\ T$ et $:= T\ \omega_2$

Les symboles R et op désignent respectivement un registre de la machine et un opérateur quelconques. Cette règle permet de ranger un opérande (représenté ici par ω_2) dans une mémoire temporaire : elle est employée quand l'expression ne peut être réduite du fait qu'un de ses opérandes n'est pas atomique. Notons que d'autres règles pourraient jouer le même rôle, par exemple :

$R = op\ \omega_1\ \omega_2 \gg R = op\ T\ T$ et $:= T\ \omega_2$ et $:= T\ \omega_1$.

La transformation effectuée entraînerait lors de l'exécution du programme une évaluation des opérandes de gauche à droite.

b) $R = \omega_o \gg R = T$ et $:= T\ \omega_o$

Cette règle donne la possibilité de transférer le contenu d'un registre dans une mémoire temporaire. Elle est utilisée dans le cas où l'expression examinée ne peut être réduite parce que le registre est incompatible avec le terme de l'expression. Une telle transformation peut éventuellement aboutir à un traitement ultérieur de la même expression. Pour éviter ce bouclage on introduit dans le transducteur un dispositif retenant à chaque utilisation de cette règle quelle est l'expression examinée. Si dans la suite de la réduction le transducteur doit de nouveau examiner cette même expression, il en considère la transformation comme impossible.

2. Protection des registres

Il peut arriver que des suites d'expressions soient du type :

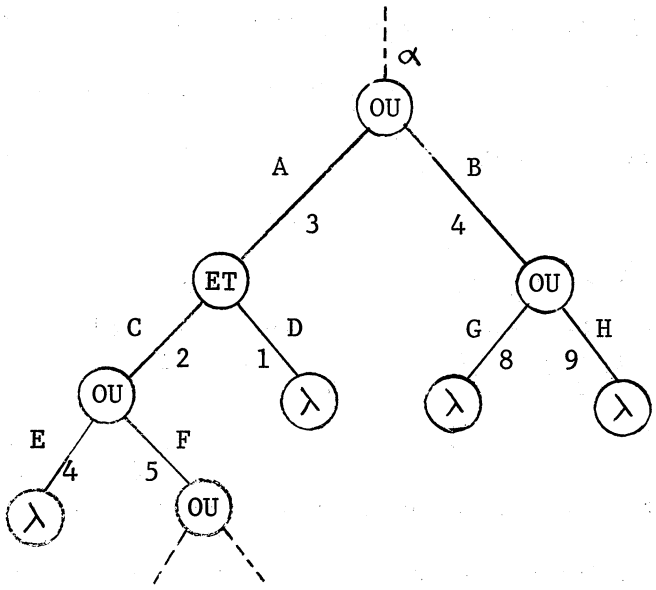
$$R_1 = \omega_1 \text{ et } R_2 = \omega_2$$

Si la réduction de l'expression désignée par ω_1 nécessite l'emploi du registre R_2, le programme objet correspondant sera erroné. Dans ce cas, il faut donc prévoir un mécanisme permettant de ranger le contenu du registre menacé dans une mémoire de travail.

VI.3.3. Optimisation

Divers critères d'optimisation peuvent être retenus pour un programme objet : rapidité d'exécution, longueur, etc. Nous avons considéré le premier et nous avons attribué à cette fin un coût à chaque instruction. Le transducteur recherche parmi les solutions possibles celles dont le coût d'instructions accumulé est minimal. Lorsque le coût de la première solution trouvée est déterminé, on abandonne les essais menant à des solutions de coût supérieur. Si un essai aboutit à un programme objet de coût inférieur, le coût de ce dernier est pris comme référence pour les comparaisons ultérieures. On remarque que dans le cas où une expression est composée d'expressions concaténées par le symbole et, le programme objet optimal correspondant est formé des programmes objet minimaux correspondant aux expressions concaténées. Cette recherche est illustrée par la figure ci-dessous. A,B,C,D,E... représentent des séquences d'instructions. Le noeud "ou" signifie que les séquences qui en sont issues produisent le même effet.

Le noeud "et" indique que les séquences suivantes doivent être juxtaposées.



La figure montre les programmes objets qui peuvent être générés à partir d'une expression α : aux points λ l'expression est réduite au vide. Ces programmes objets sont respectivement :

... ACDE , ... ACF ... D , ... BG ... BH.

Avec les coûts indiqués sur la figure, la solution minimale est le premier programme objet. On note que pour éliminer la branche F, il a suffi de comparer son coût à celui de la branche E.

Certaines transformations de la chaîne d'entrée fournissant une chaîne équivalente pourraient aussi conduire à une optimisation du programme objet. Ces transformations, représentées par des règles analogues à celles décrites précédemment, seraient effectuées avant le traitement de la chaîne donnée par le transducteur.

Exemple : la règle $- \omega_1 + \omega_2\ \omega_3 \gg -- \omega_1\ \omega_2\ \omega_3$ peut s'appliquer à la chaîne A-(B+C). Celle-ci devient A-B-C. On évite ainsi, dans certains cas, l'emploi d'une mémoire temporaire.

VI.3.4. Remarques finales

Le programme de simulation du transducteur a été écrit en Lisp-Algol par Laurent TRILLING et décrit en [Tr]. Le transducteur présente, à notre avis, deux aspects intéressants :

- les instructions du calculateur pour lequel on désire générer du code sont explicitement décrites.
- la recherche exhaustive de toutes les solutions correctes et l'introduction d'un coût associé à chaque instruction permettent de déterminer les solutions de coût minimal.

Les principales limitations actuelles de ce transducteur sont :

- l'absence d'un symbolisme indiquant facilement que l'occupation ultérieure d'un registre est interdite.
- l'absence d'un moyen permettant de générer des règles de réécriture, par exemple à la déclaration d'une procédure.

Remarquons aussi que, du fait de la grande généralité de ce transducteur, le temps nécessaire à une traduction est évidemment plus élevé que si l'on emploie une méthode du genre de celle de GRAHAM [Gra]. Plusieurs voies sont possibles pour poursuivre cette étude :

- recherche de critères permettant, au vu des règles définissant les instructions de la machine, d'éliminer des essais infructueux.

- développement de méthodes heuristiques afin d'examiner en premier lieu les branches les moins coûteuses. On pourrait, par exemple, poursuivre "simultanément" la recherche de toutes les solutions et éliminer suivant certains critères les branches paraissant mener à des solutions trop coûteuses. On pourra consulter à ce sujet l'ouvrage de SLAGLE [S1].
- élaboration sur la base de ce transducteur d'un langage de génération similaire à celui de GRAHAM : la définition de la machine serait alors donnée explicitement.

VII. Mise en oeuvre d'un système complet

> "Ne croyez surtout pas que vous aurez fini de vous amuser avec votre Meccano lorsque vous aurez construit tous les modèles décrits dans ce livre. Au contraire, c'est alors que commence la grande Aventure."
>
> (Meccano, Manuel d'instructions pour boîtes 2 et 3)

Nous décrivons ici la façon de coordonner l'utilisation des langages décrits aux chapitres précédents. Nous présentons d'abord un assemblage minimal permettant après un travail relativement réduit l'emploi combiné des langages d'édition, d'analyse et de génération. Nous décrivons ensuite une version idéale du système, qui pourra être mise à la disposition de l'utilisateur en mode conversationnel.

VII.1. Système minimal

La liaison entre les trois langages d'édition, d'analyse et de génération se fait au moyen d'appels de procédures spéciales qui provoquent le rangement de certaines tables ainsi que des noyaux interprétatifs ; ces derniers sont le résultat de la compilation des programmes écrits dans ces trois langages. Ainsi la dernière instruction d'un programme d'édition sera :

RANGER EDITION (<nom du langage>,
<numéro du passage>,
<support extérieur>)

Le <nom du langage> et le <numéro de passage> permettent de repérer le langage pour lequel on réalise un ou plusieurs passages d'édition. Le but de la procédure RANGER EDITION est d'abord de préparer le programme noyau de l'éditeur, qu'elle garnit de tous les magasins tables et variables utilisés par le programme écrit en langage d'édition ; elle le range ensuite sur un support extérieur>, dénoté par une adresse sur disques, tambours ou bandes magnétiques. Cette adresse permettra, comme nous le verrons plus loin, de charger l'éditeur en mémoire rapide quand on voudra éditer un programme écrit dans le langage défini par l'utilisateur.

Lorsque l'édition se fait en plusieurs passages, la procédure de rangement doit permettre, lors de l'exécution, d'enchaîner ces passages en ne gardant qu'une seule copie du noyau interprétatif de l'éditeur.

De façon analogue

RANGER ANALYSE (<nom du langage>,
<support extérieur>)

et RANGER GENERATION (<nom du langage>,
<support extérieur>)

utilisées comme instructions finales des programmes d'analyse et de génération, permettent le rangement des tables construites par les compilateurs de ces programmes sur les supports extérieurs donnés comme deuxièmes paramètres. Ces deux derniers appels servent à préparer les listes des points de génération, c'est-à-dire les points de programme G définis aux § IV.5 et § VI.2.

L'assemblage des trois programmes se fait au moyen d'un appel de la procédure

PREPARER COMPILATEUR (<nom du langage>,
<liste de supports extérieurs d'édition> ,
<support extérieur d'analyse>,
<support extérieur de génération>)

Cet appel joue le rôle d'appels à la fois à un éditeur de liaison et à un chargeur : à un éditeur de liaison puisque le couplage des points de génération entre les programmes d'analyse et de génération doit se faire à ce moment ; à un chargeur parce que les adresses relatives calculées lors de la compilation de ces programmes doivent être remplacées par des adresses absolues ; ceci est suivi d'un chargement en mémoire rapide du compilateur que l'on vient de construire. La réalisation pratique de cette dernière procédure pourra être complexe si

l'on doit pouvoir traiter des programmes de taille considérable écrits dans le langage défini par l'utilisateur. La subdivision du compilateur généré en plusieurs maillons sera alors faite par un spécialiste du système de la machine pour laquelle on écrira les compilateurs des trois langages proposés ici.

VII.2. Système idéal

Afin de décrire le système que nous aimerions avoir, nous présentons dans le style de Jules Verne un dialogue qui permet de dépeindre ce que pourra être la mise au point d'un compilateur. L'Homme et la Machine sont les personnages de ce dialogue.

Homme. Voici la grammaire G285 :
(il l'écrit)

1 Donnez-moi la liste des terminaux et des non-terminaux.

Machine. Voici ...
(... quelques erreurs de frappe sont détectées et des changements sont apportés à G285 ...)

2 Homme. Construisez la matrice des doublets permis.

Machine. Voici ...

3 Homme. Essayez de trouver une grammaire d'états finis équivalente à G285.

Machine. En combien de temps ?

Homme. 2 minutes.

(2 minutes après)

Machine. Je n'y arrive pas ; dois-je essayer encore ?

Homme. Oui, encore 2 minutes.

...

Machine. Je suis désolée. Je n'y arrive pas.

4 Homme. Voyez si G285 génère un LR (k) langage pour k=2.

Machine. Non, je regrette.

Homme. Essayez avec k=3.

Machine. Voici la table d'analyse :
...

Homme. Préparez-moi l'analyseur du type k=3 pour G285.
J'essaie l'analyse de quelques chaînes données.

Machine. Résultat sur l'écran.

5 Homme. Transformez le résultat en dépendance.

Machine. Résultat sur l'écran.

Homme. Je désire introduire quelques points de génération.
(... il introduit alors certains de ces points dans G285 ...)

Le dialogue se poursuit par l'utilisation du langage de génération. Evidemment les trois langages proposés dans ce travail seront utilisables. Un langage général pour la manipulation des listes (par exemple Lisp-Algol) sera lui aussi disponible. Les programmes permettant la réponse aux questions 1 et 2 sont déjà disponibles et décrits en [Mar]. On y trouvera aussi un programme qui permet la transformation d'une grammaire écrite sous FNB et contenant le symbole vide en une autre équivalente qui ne la contient pas.

L'algorithme qui permet de traiter la question 3 a été présenté au § III.5. Les résultats fournis par la machine aux questions 2 et 3 doivent aider l'utilisateur à construire la table d'états de son programme d'édition, s'il le juge nécessaire.

Le programme permettant de traiter la question 4 a été écrit en Lisp-Algol ; il est présenté en [Cou] selon l'algorithme proposé

en $|Kn|$. La question 5 est facilement traitée au moyen du programme présenté au § V.5.4.

Le dialogue décrit ci-dessus soulève le problème de la détection des erreurs. Dans notre cas, il se pose à deux niveaux : le premier se situe à l'intérieur du langage créé pour les clients de l'utilisateur de nos langages. La détection précise des erreurs des clients est d'une importance capitale pour la réussite du compilateur. Le langage d'édition a été conçu pour détecter une grande partie de ces erreurs. L'analyseur syntaxique inclu dans le langage d'analyse peut détecter un certain nombre d'erreurs, dont la position est parfois difficile à préciser. C'est même la théorie des langages qui nous rappelle la difficulté de ce problème : le complément d'un langage C.F. n'étant pas nécessairement du type C.F. $|Gi\ 2|$, il faudra utiliser un mécanisme plus puissant qu'un automate à pile pour reconnaître le complément du langage, c'est-à-dire le langage avec ses erreurs. C'est pour cette raison que nous suggérons à l'utilisateur de nos langages de signaler les erreurs de son client dans une phase d'édition. Remarquons tout de même que le langage du § VI.2 permet la détection de certains erreurs au moment de la génération.

Le deuxième niveau de reconnaissance d'erreurs concerne les moyens de détection que nous devons offrir à l'utilisateur de notre système. Ce niveau, nous le pensons, a une importance secondaire vis-à-vis du premier : l'utilisateur de nos langages étant un programmeur

avancé, il aura naturellement moins besoin d'une détection précise de ses erreurs. Néanmoins il est indispensable qu'une quantité même réduite d'erreurs soit signalée à l'utilisateur de nos langages. En ce qui concerne le langage d'édition, étant donné que son compilateur a été écrit dans son propre langage, la détection d'erreurs est facile. L'autocompilation suggérée au chapitre VIII permettra d'introduire autant de messages d'erreurs que le constructeur des compilateurs des langages d'analyse et de génération le jugera nécessaire.

VIII. Conclusions

"Il s'agit d'un simple ordinateur capable de trier rapidement un grand nombre de données sur le vocabulaire, le style, la pensée de textes qu'on lui soumet, puis de les comparer aux données qu'on a préalablement placées dans sa mémoire et à celles qu'il a recueillies au cours de ses expériences successives ... "

(R. Escarpit, Le Littératron)

"Entre lou fa e lou di
I'a tres lègo de cami."

(Entre le dire et le faire
Il y a trois lieues de chemin.")

(Proverbe provençal)

Ayant eu pour premier but la réalisation de langages d'écriture de compilateurs classiques, nous fûmes amenés en réalité à élaborer un système assez puissant qui permît la résolution de toute une gamme de problèmes liés à la compilation. Ceci ne doit pas nous surprendre : la compilation étant une opération complexe, il a fallu - afin de sauvegarder la généralité - prévoir tout un arsenal de calcul analogue à celui qui sert dans les manipulations de symboles, où la syntaxe joue un rôle prépondérant. Citons trois de ces problèmes dont l'intérêt est certain :

a. Traduction des langages de niveau élevé

Nous pensons que le système décrit dans les chapitres précédents permet la traduction de langages de niveau élevé. Ainsi Fortran pourra être traduit en Algol ; l'opération inverse, qui ne nous semble pas facile, est néanmoins réalisable. On trouvera en [Br] des indications sur la réalisation de telles traductions.

Les langages que nous avons proposés nous permettent en réalité d'aller beaucoup plus loin : l'inclusion dont nous avons parlé au chapitre I acquiert un nouvel aspect lorsqu'on essaie de traduire un langage - ayant sa propre syntaxe - en un ensemble d'appels de procédures d'un langage récepteur. Comme exemple d'une telle application, on peut réaliser un langage de traitement des problèmes matriciels ; il se traduira en un programme Algol formé d'appels à des procédures couramment utilisées dans le calcul matriciel ; un autre exemple sera la réalisation d'un langage du type Formac [Sa] qui sera traduit en Lisp-Algol. Signalons que c'est en fait tout un mécanisme de "macro-traitement" qui est à la disposition de l'utilisateur de nos langages. Son travail sera plus ou moins aisé suivant la facilité avec laquelle il pourra exprimer les transformations voulues sous la forme syntaxique requise par les langages des chapitres IV, V et VI.

b. Autocompilation ("bootstrapping")

Les langages d'édition, analyse et génération pourront certainement être perfectionnés au moyen de leur propre ensemble. On peut prévoir une amélioration de l'efficacité si l'on utilise le langage d'édition pour la phase d'édition des langages d'analyse et de génération. Remarquons à cet effet que le langage d'analyse, grâce à l'absence d'imbrications, est analysable par l'automate du langage d'édition. Il est fort possible que toutes les mises en table demandées par l'analyseur du langage d'analyse puissent être réalisées au moyen de ce même langage. Les définitions syntaxiques du § IV.5 et du § VI.2 seront alors utilisables pour réaliser l'auto-compilation.

Nous avons déjà indiqué au chapitre III que le compilateur du langage d'édition peut lui-même être décrit en langage d'édition. De même il nous paraît important de décrire les compilateurs des langages d'analyse et de génération en utilisant les trois langages que nous proposons, ne serait-ce que pour préciser le fonctionnement du compilateur d'une façon rigoureuse.

c. Traitement de problèmes en langue naturelle

L'attitude adoptée par certains spécialistes en compilation, qui est de vouloir rapprocher leurs langages des langues naturelles, a suscité et suscite encore des polémiques. Par exemple on ne peut pas dire que les efforts du groupe Cobol dans cette

direction aient été vraiment couronnés de succès. Cependant nous pensons que l'insuffisance de Cobol dans cette voie ne doit pas être prise au tragique. Dans le domaine de la recherche automatique des documents ("information retrieval") et dans d'autres où les terminaux sont accessibles aux non spécialistes, il est certain que le langage idéal ne s'éloignera pas trop des langues naturelles. Ceci n'éliminera certainement pas l'emploi par les spécialistes de langages très concis où prédomine le symbolisme mathématique.

Ne serait-ce que pour aider à la compréhension du mécanisme de génération, à la définition de l'attitude à adopter devant les problèmes mal formulés, à l'expansion du vocabulaire par apprentissage, l'étude du traitement des sous-ensembles des langues naturelles n'est pas dépourvue d'intérêt pratique.

Comme exemple d'une telle application, citons la résolution par calculateur de problèmes d'arithmétique scolaires, étudiée par BOBROW en [Bob 2]. Ces problèmes sont présentés à la machine en anglais tels qu'ils apparaissent dans les livres d'arithmétique des classes de 6-ème ou de 5-ème. Le programme écrit par BOBROW transforme d'abord la chaîne donnée en un système d'équations qui est ensuite résolu par la machine. BOBROW n'utilise pas un analyseur classique : la reconnaissance des mots et du "sens" des ensembles de mots se fait par des comparaisons successives avec une liste de mots-clefs préala-

blement rangée en mémoire et à laquelle on peut en ajouter d'autres. Il est intéressant de remarquer que l'absence de hauts niveaux d'imbrication dans les langues naturelles a conduit BOBROW, d'abord à inclure dans Lisp un langage du genre de Comit, et ensuite à l'utiliser pour l'écriture de son programme. On pourra avec nos langages aborder d'autres projets analogues à celui de BOBROW mais pour lesquels on essayera d'utiliser directement la syntaxe. Par exemple, des sous-ensembles de problèmes élémentaires de physique ou de géométrie constituent sans doute des projets intéressants, pourvu qu'ils aient à traiter un sujet et un vocabulaire bien délimités.

D'autres problèmes où la syntaxe peut être appelée à jouer un rôle important peuvent également se résoudre au moyen du système que nous avons proposé. La dérivation analytique des expressions algébriques ou trigonométriques se fait facilement au moyen des langages d'analyse et de génération. SCHORR a d'ailleurs déjà suivi cette voie par l'emploi du formalisme d'IRONS [Sc].

Revenons à la réalisation de compilateurs classiques. Quelques remarques s'imposent. Précisons d'abord un cas pour lequel la construction du système proposé ici sera indiscutablement avantageuse : si l'on doit construire plusieurs compilateurs pour une même machine de taille considérable, les noyaux de programmes que nous proposons

permettent, à l'aide de tables représentant les différents compilateurs, une mise en facteur de leurs caractéristiques communes, ce qui implique une économie de mémoire. Ceci sera encore plus intéressant dans le cas de systèmes à partage de temps où plusieurs compilateurs sollicités simultanément doivent coexister dans une zone limitée de la mémoire rapide.

La plus grande lacune existant actuellement dans la paramétrisation des compilateurs est l'absence d'un formalisme de description pratique des machines, qui permettrait la génération automatique de programmes pour plusieurs calculateurs. Nous espérons que la voie proposée au § VI.3 sera un premier pas pour combler cette lacune.

Bien que convaincu nous-même des avantages et vertus du système proposé ici, nous nous rendons bien compte qu'il ne constitue pas une panacée pour les problèmes de compilation. Le retour aux procédés artisanaux de compilation sera inévitable lorsqu'on devra traiter les cas particuliers à un langage donné ; un exemple suffit pour illustrer notre point de vue : le récupérateur d'un langage de listes construit à l'aide de notre système ne pourra être inclu dans le langage que par intervention humaine. Il est donc possible que l'utilisation de notre système devienne complexe et dépende de la machine pour laquelle il aura été construit.

Malgré ces inconvénients (et même quelques incertitudes), nous croyons que la réalisation du système proposé ici, son utilisation

et son perfectionnement au cours d'utilisations répétées constituent le seul moyen de répondre à la question : la syntaxe, telle que nous l'utilisons aujourd'hui, aidera-t-elle à établir un lien satisfaisant entre les suites de symboles représentant une commande et la réalisation de cette commande ?

APPENDICE

"Plus on dispose d'espace pour exécuter les programmes, plus les programmes augmentent pour utiliser l'espace disponible."

(1ère Loi de Parkinson, appliquée aux calculateurs)

Partie A. Utilisation des mémoires secondaires

La procédure GESTION MEMOIRE donnée en fin de cette partie permet d'étendre l'utilisation des procédures Lisp aux listes emmagasinées dans des mémoires plus lentes (disques ou tambours). Ce procédé de gestion a été inspiré de celui de l'ordinateur Atlas et du Compilateur Gier [Ki , Na 3].

La mémoire lente est divisée en groupes de 2 x 200 mots appelés pages. Les premiers de ces 200 mots correspondent à TCAR, les seconds à TCDR. Cette procédure emploie un processus rudimentaire d'apprentissage, qui laisse en mémoire rapide les pages appelées le plus souvent. Lorsqu'on a besoin d'une nouvelle page en disque, on renvoie en mémoire secondaire la page en mémoire rapide ayant eu la plus longue période d'inactivité. GESTION MEMOIRE a quatre paramètres, dont deux, BCAR et PD, sont de type booléen, et dont les deux autres sont de type entier (U et X). Les valeurs possibles de BCAR et PD sont données dans la Table 1, qui indique les affectations auxquelles elles correspondent.

PD (partie droite)

BCAR		vrai	faux
	vrai	X := TCAR [U]	TCAR [U] := X
	faux	X := TCDR [U]	TCDR [U] := X

Table 1

L'intérêt de GESTION MEMOIRE est que U peut atteindre des valeurs de l'ordre de 800.000, ce qui représente un facteur d'accroissement de 100 pour la mémoire disponible. Cet accroissement en espace de travail coûte évidemment du temps. Quelques résultats sur l'augmentation du temps d'exécution seront donnés plus loin.

Définissons d'abord les variables utilisées par GESTION MEMOIRE :

a) Variables de type entier

NPMR	Indique le nombre de positions en mémoire rapide. On définit le terme position comme l'espace en mémoire rapide pouvant emmagasiner une page quelconque de la mémoire lente.
NTP	Représente le nombre total de pages disponibles sur les disques.
POSITION LIBRE	Représente la position libre, c'est-à-dire celle où la prochaine page en provenance des disques sera rangée.
infini	Contient le plus grand entier représentable en machine.
TEMPS D'APPEL [1:NPMR]	Tableau dont le i-ème élément indique le temps d'appel de la page en i-ème position. Ce temps est donné par une horloge automatique, à chaque appel de l'indicateur de fonction "temps".

NUMERO DE PAGE [1:NPMR]	Tableau dont le i-ème élément est le numéro de la page en i-ème position.
POSITION 1:NTP]	Tableau dont le i-ème élément indique la position de la i-ème page. Quand POSITION [i] = 0, la i-ème page n'est pas en mémoire rapide.

Les définitions données ci-dessus impliquent que :

$$0 \leqslant \text{POSITION}\,[i] \leqslant \text{NPMR}\ ,$$

$$1 \leqslant \text{NUMERO DE PAGE}\,[i] \leqslant \text{NTP}$$

et NUMERO DE PAGE [POSITION [i]] = i si POSITION [i] > 0 .

b) Variables de type booléen

NON TERMINE	Indicateur de fin de lecture des disques.
ECRITURE [1:NPMR]	Table dont le i-ème élément indique si une nouvelle information a été écrite dans un mot de la page en position i.

La procédure GESTION MEMOIRE utilise deux procédures en langage-machine, qui servent à transférer les pages entre les disques et la mémoire rapide. Ce sont :

LIRE DISQUE (i,j,b)

et ECRIRE DISQUE (i,j)

où i indique la i-ème position, j la j-ème page, et où b est une

variable de type booléen qui prend la valeur vrai à l'entrée de LIRE DISQUE et la valeur faux quand la lecture est terminée. Le programme de détermination de la page la moins utilisée se déroule donc en simultanéité avec le transfert d'information de LIRE DISQUE. L'instruction étiquetée ATTENTE produit un arrêt du programme jusqu'à ce que le transfert soit terminé.

Il existe une deuxième version des procédures Lisp, dans laquelle on a remplacé chaque occurence d'une affectation de la forme indiquée en Table 1 par l'appel correspondant de GESTION MEMOIRE.

Cette deuxième version a servi à la résolution de plusieurs problèmes sur le 7044 de l'Université de Grenoble, qui comporte une unité de disques. Le temps moyen d'accès des disques est environ 10^4 fois plus lent que celui de la mémoire rapide.

Pour la plupart des problèmes que nous avons essayés, le facteur moyen de ralentissement d'exécution dû à l'utilisation des disques a rarement dépassé 3. Ce facteur augmente considérablement lorsqu'on dispose d'un nombre réduit de pages en mémoire rapide, sa valeur dépendant du problème à exécuter et de la configuration de la liste libre. On trouvera dans les Figures 1 et 2 les temps d'exécution de deux programmes caractéristiques de manipulation de symboles. On cherche dans le premier programme toutes les permutations de n objets, en gardant tous les résultats en mémoire. La Figure 1 montre la variation du temps d'exécution selon le nombre de positions disponibles

en mémoire rapide, pour n = 5 et 6. La ligne pointillée indique le temps d'exécution du même programme avec n = 5, quand on utilise seulement la mémoire rapide. Le cas où n = 6 n'a pu être résolu avec uniquement la mémoire rapide.

Le deuxième programme d'essai calcule les coefficients du développement en série de Taylor de fonctions algébriques et trigonométriques données analytiquement. Les résultats présentés en Figure 2 se rapportent à la fonction $\sin x + \cos x$ pour laquelle on a demandé 15 coefficients. La ligne pointillée de la Figure 2 indique elle aussi le temps d'exécution du programme sans mémoires secondaires.

La partie horizontale des courbes des Figures 1 et 2 montre qu'il y a un nombre critique de pages en mémoire rapide, au-dessus duquel il n'a pas d'amélioration sensible du temps de calcul. Ces résultats sont proches de ceux obtenus par NAUR pour les programmes générés par le compilateur Algol Gier [Na 3].

On trouve en Figure 3 la distribution de pages en mémoire rapide pour le problème de permutation de 6 objets, avec 10 positions en mémoire rapide. Les abscisses représentent les pages numérotées de 1 à 77. Les chiffres imprimés en ordonnées représentent le temps écoulé depuis le début de l'exécution du programme. Chaque ligne indique qu'une nouvelle page a été appelée depuis la mémoire secondaire. Les points correspondent aux pages en mémoire lente et les V aux pages en mémoire rapide. Cette figure explique les résultats obtenus, surprenants

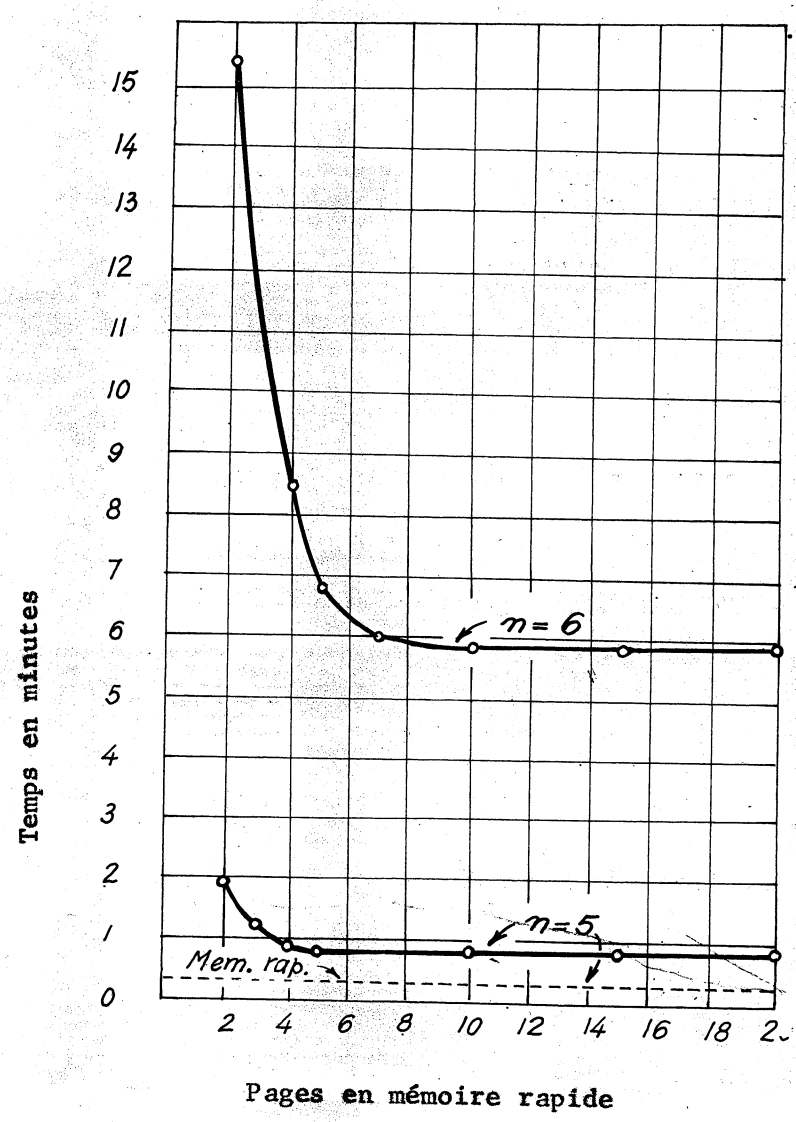


Figure 1

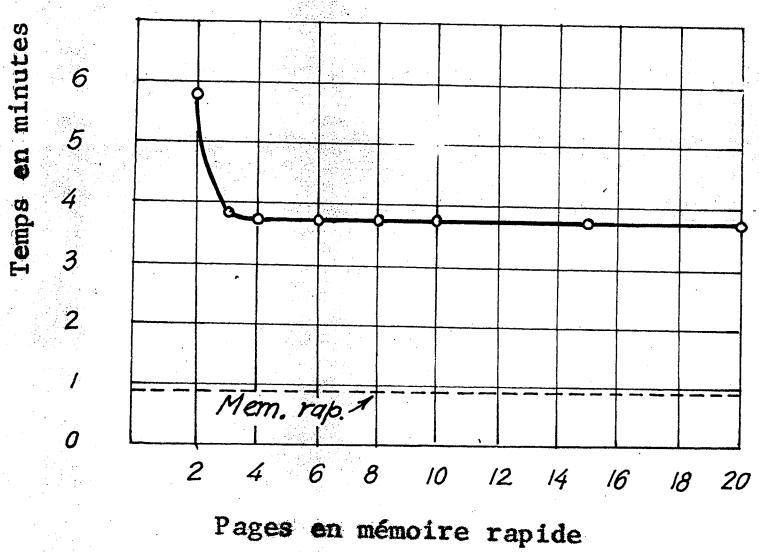


Figure 2

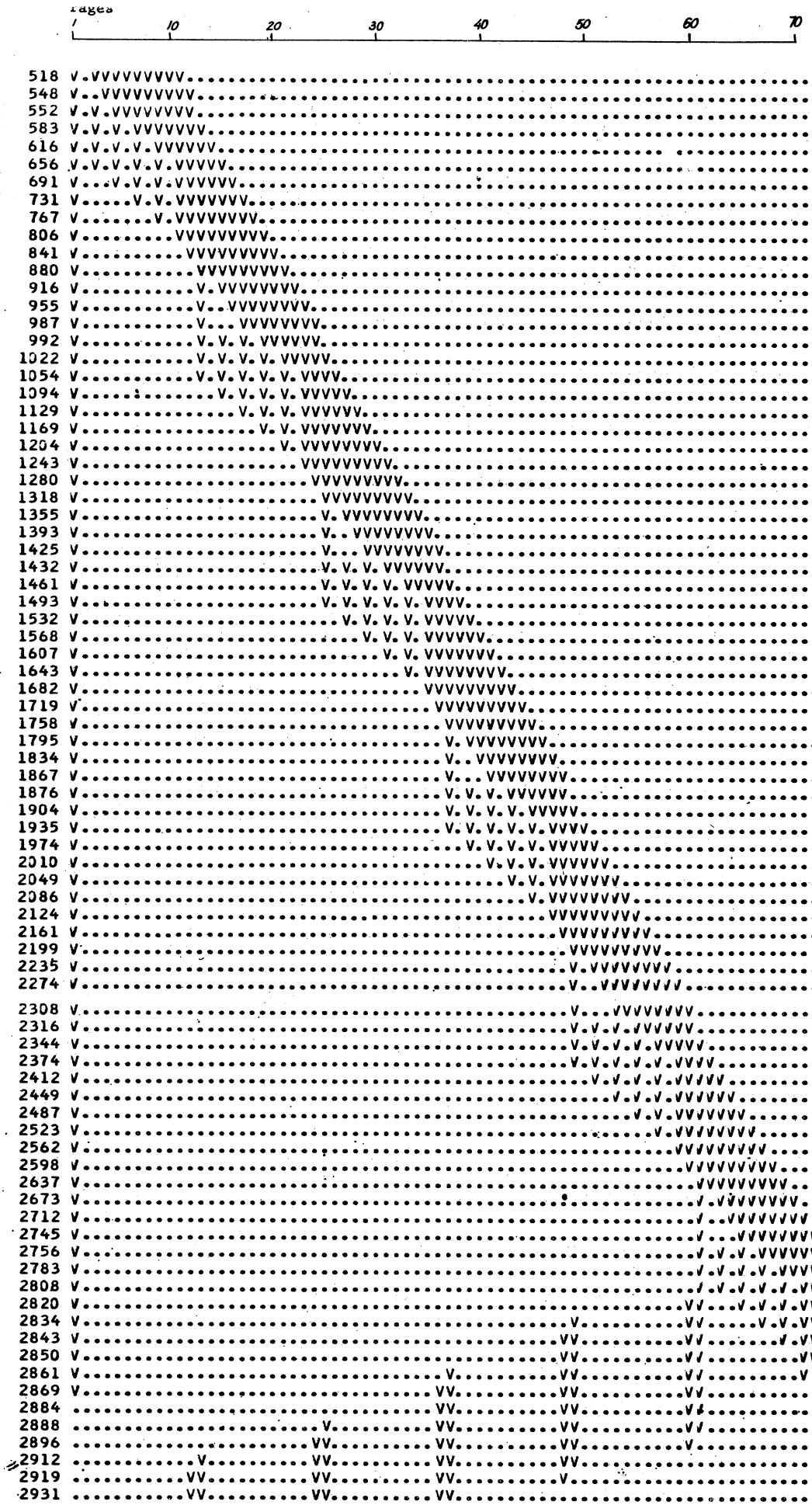


Figure 2

si l'on tient compte de la différence considérable entre les temps d'accès aux deux mémoires. La plupart des programmes contiennent des boucles qui utilisent des zones contiguës de la mémoire. L'intérêt de GESTION MEMOIRE est qu'elle se rend compte de la tendance des pages contenant ces zones contiguës à rester en mémoire.

Après avoir réalisé les procédures Lisp utilisant GESTION MEMOIRE, nous avions déjà à essayer plusieurs programmes de taille considérable - comme conséquence de la loi de Parkinson ! L'obstacle rencontré dans l'exécution de ces programmes était la grande profondeur réclamée par la pile de récursivité, ce qui épuisait la mémoire rapide. Une solution de contournement de cet obstacle consiste à ne garder en mémoire rapide que le sommet de la pile. Nous avons observé que c'était surtout pendant la phase de marquage de la récupération que la pile de récursivité débordait. La solution suggérée plus haut est particulièrement adaptée étant donné que l'on utilise toujours des variables emmagasinées au sommet de la pile.

Une autre solution assez élégante pour contourner cette difficulté nous a été communiquée par le Professeur van der Poel : on utilise un tableau de dimension fixe m au lieu de la pile de récursivité pour garder les adresses de retour d'une liste à marquer. Ayant une place limitée pour ces adresses de retour, la procédure de marquage en train de marquer une liste de profondeur p ($p \geq m$) se souvient au maximum des m dernières, les p-m premières étant "oubliées". La

procédure de marquage doit alors être rappelée plusieurs fois, jusqu'à ce que toute la liste soit marquée. Autrement dit, le marquage se fait par des balayages répétés de moins en moins profonds. Un tel procédé de marquage ne travaille pas plus lentement que dans le cas où la liste à marquer a une profondeur supérieure à m. Ce procédé est un moyen plein d'astuce pour remplacer l'espace disponible en mémoire rapide par du temps d'exécution.

Quelques remarques s'imposent sur le choix de la position la plus inactive en mémoire rapide. Nous avons essayé une deuxième stratégie de choix qui consistait à prendre une moyenne pondérée entre le temps d'inactivité t des différentes pages et le nombre d'appels n fait à ces pages. La valeur de cette moyenne pondérée (c'est-à-dire $\alpha t + \beta n$) était utilisée pour choisir la position à libérer. L'essai de plusieurs valeurs de α et β a montré que l'on obtenait un temps d'exécution minimum lorsque β s'approchait de zéro. Ces résultats montrent que le temps d'inactivité est la variable la plus importante pour le choix de la position à libérer et que peut-être restent à étudier des stratégies beaucoup plus complexes qu'une simple moyenne pondérée. Le choix d'une stratégie optimale repose évidemment sur la différence de temps d'accès entre les deux mémoires. Lorsque cette différence augmente, on peut penser à des stratégies plus complexes, étant donné qu'on peut faire un choix plus élaboré de la position libre pendant le transfert de pages.

Nous avons aussi essayé de déterminer l'effet d'autres caractéristiques de GESTION MEMOIRE. La première se rapporte aux gains obtenus par la simultanéité entre le calcul de la prochaine position libre et le transfert d'une page vers la mémoire rapide. L'élimination de cette simultanéité accroît faiblement (10 %) le temps d'exécution d'un programme. Cela s'explique par la différence considérable entre les temps d'accès des deux mémoires : le temps des transferts étant assez long, et ces transferts ne se produisant que rarement à cause de la stratégie adoptée, on pouvait en effet prévoir ces faibles gains. Il est important de remarquer que la simultanéité peut devenir cruciale dans le cas d'une différence moins importante entre les temps d'accès des deux mémoires.

L'élimination de la partie de GESTION MEMOIRE qui n'écrit des pages en mémoire lente que quand elles ont été modifiées a elle aussi diminué légèrement la vitesse d'exécution (5 à 10 %). Comme dans le cas de la simultanéité, les transferts de pages sont relativement rares et les gains obtenus sont assez faibles. Cependant on peut penser que ce facteur dépend plutôt de la nature du problème résolu que de la vitesse relative des deux mémoires.

```
procedure GESTION MEMOIRE (BCAR, U, X, PD) ; booleen BCAR, PD ; entier U, X ;
    debut entier page, ligne, i, j, min, index, temp ; booleen NON TERMINE ;
          page := (U-1) ÷ 200 ;
          ligne := U - 200 x page ;
          page := page + 1 ;
          temp := POSITION [page] ;
          si temp = 0 alors
             debut
                  LIRE DISQUE (POSITION LIBRE, page, NON TERMINE) ;
                  choix de la nouvelle position libre :
                  min := TEMPS D'APPEL [1] ; j := 1 ;
                  pour i := 2 pas 1 jusqua NPMR faire
                       si min > TEMPS D'APPEL [i]
                          alors debut
                                   j := i ;
                                   min := TEMPS D'APPEL [i]
                                fin ;
                  POSITION [page] := POSITION LIBRE ;
                  POSITION [NUMERO DE PAGE [j]]:= 0 ;
                  NUMERO DE PAGE [POSITION LIBRE] := page ;
                  ECRITURE [POSITION LIBRE] := faux ;
                  TEMPS D'APPEL [j] := infini ;
                  temp := POSITION LIBRE ; POSITION LIBRE := j ;
                  ATTENTE :
                  si NON TERMINE alors allera ATTENTE ;
                  si ECRITURE [j] alors ECRIRE DISQUE (j, NUMERO DE PAGE [j])
             fin  ;
          TEMPS D'APPEL [temp] := temps ;
          index := (temp - 1) x 200 + ligne ;
          si PD alors X := si BCAR alors TCAR [index] sinon TCDR [index]
               sinon debut
                         ECRITURE [temp] := vrai ;
                         si BCAR alors TCAR [index] := X sinon TCDR [index]:=X
                     fin
    fin procedure GESTION MEMOIRE ;
```

Partie B. Remarques sur la récupération quand on utilise les mémoires de masse

Nous étudierons ici les paramètres mis en jeu dans la récupération des listes inutiles quand on utilise des mémoires secondaires. Nous avons vu aux § V.1 et V.2 que cette récupération des listes inutiles est d'importance capitale dans la construction de compilateurs pour les langages de traitement des listes ; sa réalisation dans le cas où les listes sont rangées en mémoire secondaire (disques, tambours) est un sujet encore inexploré.

Les résultats encourageants obtenus par l'utilisation du système de gestion décrit en Partie A sont dûs au fait que les programmes contiennent des boucles qui utilisent fréquemment des zones contiguës de la mémoire. Les pages contenant ces zones contiguës auront donc tendance à rester en mémoire rapide, réduisant ainsi le nombre de transferts coûteux entre les deux mémoires. Les essais utilisant cette technique ont montré les résultats suivants : (1) les temps d'exécution augmentent brusquement lorsqu'on dispose d'un nombre réduit de pages en mémoire rapide ; (2) pour un programme donné il existe un nombre minimum de pages en mémoire rapide au-dessus duquel on n'obtient pas de réduction sensible des temps d'exécution.

En essayant de combiner ces résultats avec les techniques de récupération que nous avons décrites aux § V.1 et V.2, nous avons

formulé l'hypothèse qu'après quelques appels de GC1 la liste libre contiendrait des mots très dispersés dans la mémoire, ce qui pourrait augmenter considérablement la durée du calcul ultérieur. Si cette hypothèse se confirmait il serait intéressant de déterminer dans quelles conditions GC2 est plus efficace que GC1.

Les paramètres suivants interviennent dans la vitesse de récupération :

1. Géométrie des listes

La géométrie des listes est un paramètre important, dont la variation est difficile à exprimer quantitativement. Pour définir la géométrie des listes nous avons choisi les paramètres suivants :

a. profondeur ;

b. longueur ;

c. degré d'imbrication avec d'autres listes.

Si l'on représente par L [i] et L [j] deux listes différentes et par A un atome, les constructions associées aux items a, b et c sont respectivement :

L [i] := cons (L [i], nil) ;

L [i] := cons (A, L [i]) ;

L [i] := cons (élément de (L [j]), L [i]) ;

Afin d'introduire ces paramètres dans les programmes d'essai, on peut affecter des probabilités d'apparition à chacune des construction ci-dessus et choisir leur enchaînement par la méthode de Monte-Carlo. On utilise aussi des tirages aléatoires pour déterminer "i", "j" et "l'élément de (L [j])".

2. Evolution des calculs

La variation de ce paramètre est encore plus difficile à estimer que pour le précédent. Son introduction dans les programmes d'essai peut se faire par l'exécution d'un même problème en partant de conditions initiales différentes de la liste libre (par exemple après des nombres différents de récupérations).

3. Nombre de pages en mémoire rapide

Le nombre de pages en mémoire rapide est un paramètre important dans un système de gestion du type de l'ordinateur Atlas. Il est vraisemblable que le calcul se déroule plus rapidement après la récupération avec retassement qu'après GC1, surtout si l'on dispose d'un nombre réduit de pages en mémoire rapide.

4. Nombre de pages en mémoires secondaires

Il est possible que pour certains problèmes on puisse éviter la récupération pourvu qu'il existe suffisamment de mémoires. D'un autre côté il est encore possible que la récupération soit nécessaire pour réduire le nombre de transferts de pages, surtout dans le cas où l'on construit des listes à partir d'éléments très dispersés dans la mémoire lente. Cependant la récupération reste le seul espoir de solution d'un problème lorsqu'on dispose d'un nombre réduit de pages en mémoire lente.

La taille d'une page peut elle aussi être considérée comme une des variables en jeu dans la récupération. Cependant dans notre cas, et probablement dans la majorité des cas, cette taille est fixée par la structure de la machine.

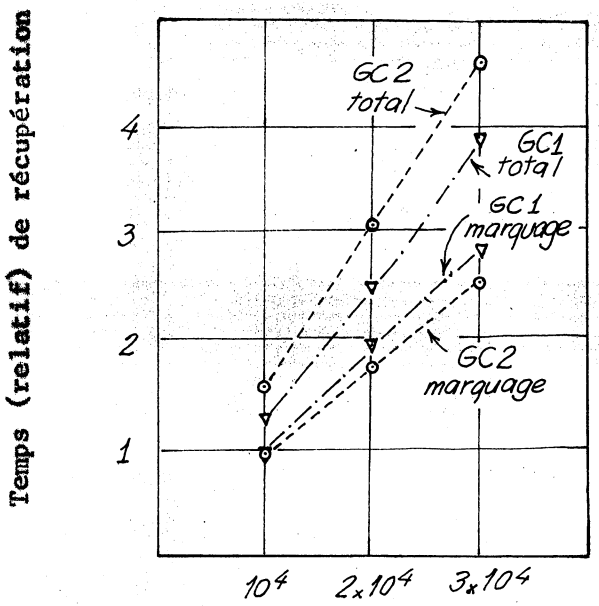
Bien que l'on puisse imaginer des expériences portant sur les paramètres ci-dessus, la nature des deux premiers rend difficile une comparaison entre les résultats de ces expériences et les problèmes réels.

Dans les essais en nombre limité que nous avons réalisés, nous avons placé l'accent sur l'étude des paramètres 3 et 4. Nous avons choisi la méthode de Monte-Carlo pour générer deux listes, de la façon suggérée plus haut. Lorsque l'encombrement des constructions atteignait un certain nombre de mémoires (10^4, 2×10^4, 3×10^4 mots), on appelait le programme de récupération pour effacer une des listes. Le temps moyen de marquage et le temps total de récupération sont présentés en Figure 4. Ces résultats correspondent au cas où il y avait 15 pages en mémoire rapide. Nous avons obtenu des résultats presque identiques en traitant le même problème avec 5 et 10 pages en mémoire rapide. Cela indique que pour ce programme le nombre critique* de pages en mémoire rapide n'est pas supérieur à 5.

La Figure 4 montre que les temps de récupération par GC2 sont toujours supérieurs à ceux de GC1 ; les temps de marquage de GC2 sont en revanche toujours inférieurs à ceux de GC1. Ces résultats étaient bien prévisibles : en effet, après un certain nombre de récupérations par GC2, les listes utiles occupent des mots adjacents

---

(*) C'est-à-dire, le nombre de pages au-dessus duquel il n'y a pas de réduction possible dans les temps d'exécution d'un problème.



15 pages disponibles
en mémoire rapide

Figure 4

de la mémoire ; le marquage étant réalisé sur ces mots, il y a réduction du nombre de transferts entre les deux mémoires. Nous avons observé aussi que les temps de calcul entre deux récupérations par GC2 étaient plus petits que les temps de calcul entre deux appels de GC1. Ces résultats sont explicables par les raisons que nous venons de mentionner pour le marquage des listes utiles. A ce propos il faut aussi remarquer que l'effet de GC1 est équivalent à une réduction du nombre de pages en mémoire rapide : GC1 récupère les "trous" de mémoires occupés par les listes inutiles ; la place ainsi récupérée étant utilisée dans les nouvelles constructions, il y aura moins d'espace utile disponible au même moment en mémoire rapide.

Remarques finales

Bien que nous n'ayons réalisé qu'un nombre limité d'expériences, nous pouvons déjà dégager les résultats suivants :

a. La récupération avec retassement est évidemment plus coûteuse que la récupération sans retassement. Nous espérions que le retassement serait utile par l'aide indirecte qu'il apporterait à la réduction du nombre de transferts entre les mémoires secondaires et rapides. En fait le temps perdu par le retassement dépasse le temps gagné par la réduction du nombre de transferts. Néanmoins, il est possible que le retassement soit avantageux dans les cas où le temps de calcul est supérieur au temps total de récupération. Autrement dit le choix entre GC1 et GC2 doit être fait d'après le rapport entre la durée du calcul proprement dit et le temps total de récupération. Si ce rapport est faible le retassement n'est pas avantageux, sauf si le paragraphe b. s'applique.

b. Dans la construction des compilateurs le problème principal est de déterminer le moment opportun d'appel du programme de récupération. Si l'on dispose d'un nombre total de pages réduit, il est avantageux de repousser cet appel jusqu'au moment où la mémoire libre est épuisée. D'un autre côté, lorsque l'on dispose au total d'un grand nombre de pages et qu'il ne peut en coexister en mémoire qu'un nombre limité, la récupération avec retassement est le seul

moyen d'éviter un traitement intolérablement lent de listes ayant leurs éléments très dispersés dans la mémoire. Dans ce cas, on peut déclencher automatiquement la récupération lorsque le nombre (par unité de temps) de transferts de pages entre les deux mémoires devient plus grand qu'un nombre maximum préalablement choisi.

Parmi les projets de poursuite des travaux ci-dessus décrits il serait intéressant d'écrire un programme simulant l'action du récupérateur, plutôt que de réaliser des essais de récupération effective.

BIBLIOGRAPHIE

Nous signalons que les références [Fl 5 , BGL , GP , HSS , BC 2 , Gi 2 , Bob 1] contiennent à leur tour des bibliographies importantes.

Bac — J.W. Backus, The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference, Proc. International Conf., Information Processing, UNESCO, Paris, Juin 1959.

Bar — D.W. Barron et al., The Main Features of CPL, Computer Journal, Juillet 1963.

Bas — A.L. Bastian, Jr., A phrase-structure language translator, Air Force Cambridge Res. Labs. Report No. AFCRL-69-549, Hanscom Field, Mass., Août 1962.

BB — E. Berkeley et D. Bobrow, The Programming Language Lisp ; Its Operation and Applications, Information International, Inc., Cambridge, Mass., 1964.

BC 1 — M. Brasseur et J. Cohen, Etudes de techniques de compilation, 4e Congrès AFCALTI, Versailles, 1964 (paru aussi comme Note Technique IMAG, Description en Algol d'un compilateur Algol simplifié).

BC 2 — M. Brasseur et J. Cohen, Algorithmes d'analyse syntaxique pour langages "context-free", Chiffres, vol. 8, n° 2 et 3, 1965.

BF — M.P. Barnett et R.P. Futrelle, Syntactic analysis by digital computer, Comm. A.C.M., vol. 5, Octobre 1962.

BGL — L. Bolliet, N. Gastinel et P.J. Laurent, Un Nouveau Langage Scientifique - ALGOL - Manuel Pratique, Hermann, Paris, 1964.

BM — R.A. Brooker et D. Morris, A general translation program for phrase-structure languages, J. A.C.M., vol. 9, Janvier 1962.

BMMR — R. Brooker, I. Maccallum, D. Morris, J. Rohl, The compiler's compiler, Third Annual Review of Automatic Programming, Pergamon Press, 1963.

Bob 1 — D.G. Bobrow, Syntactical Analysis of English by Computers--a Survey, Proc. Fall Joint Computer Conference, Spartan, Baltimore, Md., 1963.

Bob 2 — D.G. Bobrow, Natural Language Input for a Computer Problem Solving System, Ph.D. Thesis, M.I.T., Septembre 1964.

Bol — L. Bolliet, Compiler Writing Techniques, Notes on a series of lectures given at the NATO Summer School of Programming Languages, Villard-de-Lans, Septembre 1966.

Bou — J.C. Boussard, Etude et Réalisation d'un compilateur Algol 60 sur calculatrice électronique du type IBM 7090/94 et 7040/44, Thèse, Université de Grenoble, Juin 1964.

BPS — Y. Bar-Hillel, M. Perles, et E. Shamir, On formal properties of simple phrase structure grammars, Z. Phonetik, Sprach. Kommunikationsforsch, vol. 14, 1961; paru dans Y. Bar-Hillel, "Language and Information", Addison-Wesley Publishing Company, Inc., Reading, Mass., 1964.

Br — H. Breuer, Dictionary for Computer Languages, A.P.I.C. Studies in Data Processing N° 6, Academic Press, 1966.

BW — D. Bobrow et J. Weizenbaum, List Processing and Extension of Language Facility by Embedding, I.E.E.E. Transactions on Electronic Computers, Août 1964.

CD — J. Cohen et Nguyen Huu Dung, Définition de procédures Lisp en Algol. Exemples d'utilisation, Chiffres, vol. 8, n° 4, 1965.

Ch 1 — N. Chomsky, On certain formal properties of grammars, Inf. and Control, vol. 2, Juin 1959 ; (addendum) A note on phrase structure grammars, Inf. and Control, vol. 2, Décembre 1959.

Ch 2 — N. Chomsky, Formal Properties of Grammars, Handbook of Mathematical Psychology, vol. 2, Wiley, New York, 1963.

Cl — H.A. Clampett, Randomized Binary Searching with Tree Structures, Comm. A.C.M., Mars 1964.

CL T. Cheatham et G. Leonard, Introduction to the CLII Programming System, Computer Associates, Inc., CA-63-7-SD, Novembre 1963.

Coh J. Cohen, Remarques et additions aux procédures Lisp en Algol, Calcolo, Fasc. IV, vol. III, 1966.

Cou J. Courtin, Etude des langages analysables de gauche à droite et applications, Thèse, Université de Grenoble, à paraître.

CS T.E. Cheatham, Jr. et K. Sattley, Syntax-directed compiling, Proc. Spring Joint Computer Conf., Spartan Books, Baltimore, Md., vol. 25, 1964.

CT J. Cohen et L. Trilling, Remarks on Garbage-Collection Using a Two-Level Storage, Nordisk Tidskrift for Informations-Behandling, Bind 6, Hefte 4, 1967.

dM 1 J. du Masle, Ecriture en Algol d'un compilateur Algol, Note Technique, Institut de Mathématiques Appliquées, Grenoble, Janvier 1965.

dM 2 J. du Masle, Ecriture d'un compilateur Algol à l'aide d'un système autocodeur, Congrès A.F.I.R.O., Versailles, Avril 1964.

DN O.J. Dahl et K. Nygaard, SIMULA - A Language for Programming and Description of Discrete Event System, Norweigian Computing Center, Norvège, Mai 1965.

Do T.A. Dolotta, Méthode d'édition d'un programme en langage symbolique, Proc. International Computation Center, Symposium on Symbolic Languages, Gordon and Breach, 1962.

Ev T.G. Evans, Machine-Aided Design of Context-Free Grammars, Air Force Cambridge Research Laboratories, AFCRL-65-747, Octobre 1965.

Fe 1 J. Feldman, A Formal Semantics for Computer Oriented Languages, Ph.D. Thesis, Carnegie Institute of Technology, Mai 1964.

Fe 2 — J. Feldman, A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler, Comm. A.C.M., Janvier 1966.

Fl 1 — R.W. Floyd, On the non-existence of a phrase structure grammar for ALGOL 60, Comm. A.C.M., vol. 5, Septembre 1962.

Fl 2 — R.W. Floyd, A descriptive language for symbol manipulation, J. A.C.M., vol. 8, Octobre 1961.

Fl 3 — R.W. Floyd, Bounded context syntactic analysis, Comm. A.C.M., vol. 7, Février 1964.

Fl 4 — R.W. Floyd, Syntactic Analysis and Operator Precedence, J. A.C.M., vol. 10, 1963.

Fl 5 — R.W. Floyd, The Syntax of Programming Languages - A Survey, I.E.E.E. Transactions on Electronic Computers, Août 1964.

Gai — H. Gaifman, Dependency Systems and Phrase-Structure Systems, Information and Control 8, 1965.

Gar 1 — J.W. Garwick, Gargoyle - A Language for Compiler Writing, Comm. A.C.M., Janvier 1964.

Gar 2 — J.W. Garwick, Data Storage in Compilers, Nordisk Tidskrift for Informations-Behandling, n° 4, 1964.

Ge — C.W. Gear, High Speed Compilation of Efficient Object Code, Comm. A.C.M., Août 1965.

Gi 1 — S. Ginsburg, An Introduction to Mathematical Machine Theory, Addison-Wesley Publishing Co., Reading, Mass., 1962.

Gi 2 — S. Ginsburg, The Mathematical Theory of Context-Free Languages, Mc Graw Hill, 1966.

GP — T.H. Griffiths et S.R. Petrick, On the Relative Efficiencies of Context-Free Grammar Recognizers, Comm. A.C.M., Mai 1965.

Gra — R. Graham, Bounded context translation, Proc. Spring Joint Computer Conf., Spartan Books, Baltimore, Md., vol. 25, 1964.

Gre 1 — S. Greibach, Inverses of Phrase Structure Generators, Mathematical Linguistics and Automatic Translation, Ph.D. Thesis, Harvard University, 1963.

Gre 2 — S. Greibach, A New Normal Form Theorem for Context-Free Phrase Structure Grammars, J. A.C.M., vol. 12, Janvier 1965.

Gre 3 — S. Greibach, Formal Parsing Systems, Comm. A.C.M., Août 1964.

Gro — M. Gross, Linguistique Mathématique et langages de programmation, Revue Française de Traitement de l'Information, Chiffres, n° 4, 1963.

Hal — M.H. Halstead, Machine-Independent Computer Programming, Spartan Books, 1962.

Har — M. Harrison, Introduction to Switching and Automata Theory, Mc Graw Hill, 1965.

Ho — C.A.R. Hoare, A proposal for record handling in Algol X, Document du groupe WG 2.1 de l'IFIPS, 1966, à paraître dans le rapport du nouveau langage Algol.

HSS — Harvard Summer School, Language Data Processing, A Special Program, The Computation Laboratory, Août 1964.

In 1 — P.Z. Ingerman, A Translation Technique for Languages . . ., Proc. of the International Computation Center, Symposium on Symbolic Languages, Gordon and Breach, 1962.

In 2 — P.Z. Ingerman, A Syntax-Oriented Translator, Academic Press, 1966.

Ir 1 — E.T. Irons, A syntax directed compiler for ALGOL 60, Comm. A.C.M., vol. 4, Janvier 1961.

Ir 2 — E.T. Irons, An error-correcting parse algorithm, Comm. A.C.M., vol. 6, Novembre 1963.

Ir 3 E.T. Irons, The structure and use of the syntax-directed compiler, Annual Review in Automatic Programming, Pergamon Press, The Macmillan Company, New York, vol. 3, 1963.

Ir 4 E.T. Irons, Maitenance Manual for PSYCO, the Princeton Syntax Compiler, Institute for Defense Analysis, Princeton, 1961.

Iv K.E. Iverson, A Programming Language, John Wiley and Sons, 1962.

Je J. Jensen, Generation of Machine Code in Algol Compilers, BIT, 5, 1965.

Joh P. Johansen, Construction of Recognition Devices for Regular Languages from their BNF Definition, Nordisk Tidskrift for Informations-Behandling, BIT, n° 4, 1966.

Jor Ph. Jorrand, Etude et réalisation d'un langage d'édition, Note technique, à paraître, Institut de Mathématiques Appliquées, Grenoble.

Ki T. Kilburn et al., One-level storage system, I.R.E. Transactions on Electronic Computers, Avril 1962.

Kn D.E. Knuth, On the Translation of Languages from Left to Right, Information and Control 8, 1965.

KO S. Kuno et A.G. Oettinger, Multiple-Path Syntactic Analyser, Mathematical Linguistics and Automatic Translation, Harvard University, 1963.

Ks R. Kurki-Suonio, On Top-to-Bottom Recognition and Left-Recursion, Comm. A.C.M., Juillet 1966.

La P.S. Landweber, Decision Problems of Phrase-Structure Grammars, I.E.E.E. Transactions on Electronic Computers, 1964.

Le B.M. Leavenworth, Fortran IV as a Syntax Language, Comm. A.C.M., Février 1964.

1P J. Le Palmec, Etude d'un langage intermédiaire pour la compilation d'Algol 60, Thèse, Université de Grenoble, Juin 1966.

LW R.S. Ledley et J.B. Wilson, Automatic-programming-language translation through syntactical analysis, Comm. A.C.M., vol. 5, Mars 1962.

Mar F. Martin, Détermination de certaines caractéristiques des grammaires et langages C.F., Thèse, Université de Grenoble, à paraître.

May B.H. Mayoh, Irons' procedure DIAGRAM, Comm. A.C.M., lettre de correction, vol. 4, Juin 1961.

Mc 1 J. Mc Carthy, Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I, Comm. A.C.M., Avril 1960.

Mc 2 J. Mc Carthy et al., LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, Mass., 1962.

Me H.H. Metcalfe, A Parameterized Compiler Based on Mechanical Linguistics, Annual Review in Automatic Programming, vol. 4, Pergamon Press, 1964.

Moo E.F. Moore, ed., Sequential Machines : Selected Papers, Addison-Wesley Publishing Co., 1964.

Mor D. Morris, The Use of Syntactic Analysis in Compilers, Introduction to System Programming, ed. P. Wegner, Academic Press, 1964.

Na 1 P. Naur et al., Report on the algorithmic language ALGOL 60, Comm. A.C.M., vol. 3, Mai 1960 ; Annual Review of Automatic Programming, Pergamon Press, The Macmillan Co., New York, 1961, vol. 2 ; Numerische Mathematik, vol. 2, Mars 1960.

Na 2 P. Naur et al., Revised report on the algorithmic language ALGOL 60, Comm. A.C.M., vol. 6, Janvier 1963 ; Numerische Mathematik, vol. 4, Mars 1963 ; Computer Journal, vol. 5, Janvier 1963.

Na 3 — P. Naur, The Design of the Gier Algol Compiler, Nordisk Tidskrift for Informations-Behandling, BIT 3, 1963.

Ndx — Nguyen Dinh Xuan, Méthodes d'analyses syntaxiques descendantes pour langages "context-free", Thèse, Université de Grenoble, Janvier 1966.

Nhd — Nguyen Huu Dung, Aspects non numériques de la programmation Algol, Thèse, Université de Grenoble, Juin 1965.

Oe — A.G. Oettinger, Automatic Syntactic Analysis and the Push-down Store, Proc. Symposia in Applied Mathematics, vol. XII, 1961.

Ol — M. Oliver, Le bricolage est un jeu d'enfants, Plon, 1966.

PIS — A.J. Perlis, R. Iturriaga, T. Standish, A preliminary sketch of Formula Algol, Carnegie Institute of Technology, Avril 1965.

RIO — Report on Input-Output for ALGOL 60, Comm. A.C.M., Octobre 1964.

Ro — S. Rosen, A Compiler-Building System Developed by Brooker and Morris, Comm. A.C.M., Juillet 1964.

RR — B. Randell et L.J. Russel, Algol 60 Implementation, A.P.I.C. Studies in Data Processing, n° 5, Academic Press, 1964.

SB — J.E. Sammet et E.R. Bond, Introduction to Formac, I.E.E.E. Transactions on Electronic Computers, Août 1964.

Sc — H. Schorr, Analytic Differentiation Using a Syntax-Directed Compiler, Computer Journal, Janvier 1965.

Sl — J.R. Slagle, A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, J. A.C.M., Octobre 1963.

SW — H. Schorr et W.M. Waite, An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, IBM Research Report RC-1450, 1965.

Tr — L. Trilling, Contribution à l'étude de la phase de génération en compilation : Application au traitement des enregistrements, Thèse, Université de Grenoble, à paraître.

TTW — W. Taylor, L. Turner, R.W. Waychoff, A Syntactical Chart of ALGOL 60, Comm. A.C.M., Septembre 1961.

Un — S.H. Unger, On Syntax Directed Translations, R.C.A. Laboratories, Private Publications, 1963.

Va — B. Vauquois, Cours de logique et programmation, Polycopié, Université de Grenoble, 1964.

VV — G. Veillon et J. Veyrunes, Etude de la réalisation pratique d'une grammaire "context-free" et de l'algorithme associé, Centre d'Etudes pour la Traduction Automatique, Section de Grenoble, G-001-1, Avril 1964.

Wa 1 — S. Warshall, A syntax-directed generator, Proc. Eastern Joint Computer Conf., Spartan Books, Baltimore, Md., vol. 20, 1961.

Wa 2 — S. Warshall, A theorem on Boolean matrices, J. A.C.M., vol. 9, Janvier 1962.

We — J. Weizenbaum, Symmetric List Processor, Comm. A.C.M., Septembre 1963.

VU

Grenoble, le

Le Président de la Thèse

VU

Grenoble, le

Le Doyen de la Faculté des Sciences

VU, et permis d'imprimer,

Le Recteur de l'Académie de GRENOBLE