Définition et implantation de la sémantique des langages de programmation

Bruce Willis

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

pour obtenir le grade de

Docteur ès sciences

par

Bruce WILLIS

Définition et implantation de la sémantique des langages de programmation

Thèse soutenue le 22 mars 1974 devant la commission d'examen

Monsieur	J. KUNTZMANN	Président
Monsieur	M. GRIFFITHS	Rapporteur
Monsieur	C.H.A. KOSTER	Examinateurs
Monsieur	G. VEILLON	

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

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

PROFESSEURS TITULAIRES

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

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

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

PROFESSEURS ASSOCIES

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

PROFESSEURS SANS CHAIRE

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

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

Mle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Dermatologie
	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Yves	Chimie
	BEGUIN Claude	Chimie organique
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BILLET Jean	Géographie
	BLIMAN Samuel	Electronique (EIE)
	BLOCH Daniel	Electrotechnique
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BOUCHET Yves	Anatomie
	BOUVARD Maurice	Mécanique des fluides
	BRODEAU François	Mathématiques (IUT B)
	BRUGEL Lucien	Energétique
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CHIBON Pierre	Biologie animale
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	DURAND Francis	Métallurgie
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROULADE Joseph	Biochimie médicale
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Médecine préventive
	IDELMAN Simon	Physiologie animale
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	JOLY Jean-René	Mathématiques Pures
	JOUBERT Jean-Claude	Physique du solide
	JULLIEN Pierre	Mathématiques Pures
	KAHANE André	Physique générale
	KUHN Gérard	Physique
	LACOUME Jean-Louis	Physique
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LANCIA Roland	Physique atomique
	LE JUNTER Noël	Electronique
	LEROY Philippe	Mathématiques
	LOISEAUX Jean-Marie	Physique nucléaire
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LUU DUC Cuong	Chimie organique
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)

MM.	MAYNARD Roger	Physique du solide
	MICHOULIER Jean	Physique (IUT A)
	MICOUD Max	Maladies infectieuses
	MOREAU René	Hydraulique (INP)
	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PERRIN Louis	Pathologie expérimentale
	PFISTER Jean-Claude	Physique du solide
	PHELIP Xavier	Rhumatologie
Mlle	RIERY Yvette	Biologie animale
MM.	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RENAUD Maurice	Chimie
	RICHARD Lucien	Botanique
Mme	RINAUDO Marquerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VAN CUTSEM Bernard	Mathématiques appliquées
	VEILLON Gérard	Mathématiques appliquées (INP)
	VIALON Pierre	Géologie
	VOOG Robert	Médecine interne
	VROUSSOS Constantin	Radiologie
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES ASSOCIES

MM.	BOUDOURIS Georges	Radioélectricité
	CHEEKE John	Thermodynamique
	GOLDSCHMIDT Hubert	Mathématiques
	SIDNEY STUARD	Mathématiques Pures
	YACOUD Mahmoud	Médecine légale

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

Mme	BERIEL Hélène	Physilogie
Mme	RENAUDET Jacqueline	Microbiologie

Fait le 30 mai 1972.

Monsieur KUNTZMANN me fait l'honneur de présider le jury de cette thèse. Je l'en remercie vivement.

Je voudrais exprimer toute ma gratitude envers M. GRIFFITHS pour ses suggestions et ses commentaires féconds.

Je suis très redevable au Professeur C.H.A. KOSTER et à Monsieur VEILLON d'une lecture attentive et critique.

Je voudrais remercier tout particulièrement Ph. CHATELIN et P.Y. CUNIN pour leurs efforts considérables de "francisation" du texte. D'autre part, il me faut dire que je dois beaucoup d'idées de cette thèse à mes collègues et amis : S. SCHUMAN, Ph. CHATELIN, P.Y. CUNIN, M. SIMONET et P. COUSOT pour n'en citer que quelques uns. J'ai eu l'heureuse fortune d'être associé à deux groupes différents à Grenoble, l'un travaillant sur les langages extensibles avec PL/1, l'autre sur le compilateur Algol 68. Nous avons travaillé longtemps ensemble aussi est-il très difficile de dissocier qui a eu l'idée de quoi ! Lorsque ce sera possible, je décrirai l'évolution de chaque idée et la contribution de chacun.

B.W.

TABLE DES MATIERES

APPENDICES

0. INTRODUCTION

Au cours de ces dernières années, j'ai eu l'occasion de participer à plusieurs projets orientés vers les compilateurs : un compilateur Fortran pour un calculateur "multiprocessing" à l'Université de Chicago, une étude des problèmes d'implémentation et de définition de PL/1 et finalement l'architecture d'un compilateur Algol 68. Ces deux derniers projets à Grenoble.

Comme beaucoup avant moi, j'ai été frappé par la grande différence qu'il existe entre la méthodologie disponible pour décrire les propriétés syntaxiques des langages de programmation et celle dont on dispose pour établir leur sémantique. Avec la venue de la notation BNF utilisée pour décrire les grammaires hors-contexte dans le rapport Algol 60, une avance rapide a été faite dans le développement des compilateurs dirigés par la syntaxe (IRONS [14], FOSTER [10] ...). Un nombre étonnant d'analyseurs ont été développés ; chacun applicable à un type de grammaire plus ou moins pratique. La notion d'équivalence entre une grammaire et la partie "reconnaissance" d'un compilateur est devenue un lieu commun. Il semble presque trivial de la mentionner ici. Tout ce bagage technologique a conduit à une compétition croissante entre les informaticiens envers les aspects syntaxiques d'un compilateur.

Si la syntaxe est bien appréhendée qu'en est-il de la sémantique ? On ne peut prétendre que les informaticiens ne sont pas conscients de ce problème. Il est apparu en effet un ensemble de techniques de plus en plus raffinées pour exprimer la sémantique des langages de programmation (Landin [16], le Langage de Vienne [25] ...). Ces formalisations quoique très utiles pour améliorer la précision des définitions du langage, donnent seulement un aperçu de la manière de les implémenter dans un compilateur : (aperçu cependant précieux). Un exemple frappant en est la révision nouvelle du rapport Algol 68 [22].

Un fossé s'est creusé entre la définition et l'implémentation de la sémantique. Par comparaison un analyseur syntaxique répond d'une manière adéquate à ces deux nécessités pour la syntaxe. Comme "implémenteur" j'ai senti avec acuité ce besoin d'appliquer directement la définition aux fonctions sémantiques d'un compilateur et cela, si possible, d'une manière automatique ou semi-automatique.

A ce point il aurait été agréable d'être capable de dire qu'une façon de combler ce fossé avait été trouvée et ceci "d'un seul coup". L'intention plus modeste de cette thèse est de construire un pont entre la définition et l'implémentation des notions sémantiques en rapportant l'expérience acquise en travaillant à des compilateurs. En d'autres termes, je souhaite montrer qu'une définition ne nécessite pas d'être impossible à implémenter directement et vice-versa.

En faisant cela, il est clair que des compromis seront à faire, à la fois pour la définition et pour l'implémentation. Les techniques de définition devront abandonner les opérateurs universels ($\exists$, $\forall$) si chères aux définitions de Vienne, puisque cela peut conduire à des recherches très longues au moment de l'exécution. Corollairement l'implémentation peut accepter des manières moins directes de faire les choses, afin que trop de cas spéciaux n'apparaissent dans la définition.

Ces avertissements en tête, il devrait être clair que la question la plus brûlante est celle-ci : "Un pont entre implémentation et définition de la sémantique est-il même possible ?" C'est la raison de l'accent porté sur l'évaluation des méthodes proposées dans cette thèse.

0.1. La méthode -

Plutôt que de produire encore une autre définition formelle de langage, l'approche à saisir est quelque peu différente.

Elle repose sur la simple observation que les phrases d'un texte peuvent être remplacées par des phrases structurellement plus simples qui sont "sémantiquement équivalentes". Comme exemple, on prendra un texte écrit dans un certain langage de programmation. Supposons un processus qui produit un nouveau texte ayant le même "effet" à l'exécution que l'original. Si le langage du nouveau texte est un sous-langage du langage original et si cette condition est vraie pour tous les textes originaux possibles, alors on peut dire que le processeur fait une analyse sémantique partielle.

Le processus de traduction est fait pour produire une détermination de la signification du texte initial. C'est la même méthode - il va de soi - qui est appliquée lorsque l'on traite un texte compliqué en langue naturelle et qu'on le réécrit en mots "plus simples", ou pour prendre un exemple extrême, lorsqu'un langage de programmation de haut niveau est traité par un compilateur et traduit en code machine équivalent.

L'idée est assez claire, mais il est nécessaire d'avoir un mécanisme adapté au problème de la détermination des phrases et de leurs substitutions. On pense immédiatement aux macros dans ce contexte mais lorsqu'on est confronté aux problèmes réels de traduction des langages de haut-niveau, il devient vite évident que n'importe quel type de macro substitution ne convient pas.

Au premier chapitre une forme particulièrement puissante de macro substitution est introduite : les macros syntaxiques. Elle sert de support pour les chapitres suivants.

Exprimé de façon succinte, les macros-syntaxiques permettent la traduction pendant l'analyse du texte d'entrée. Par conséquent, le macroprocesseur accepte un texte en langage d'entrée mais produit une analyse dans le texte équivalent écrit en langage final. L'idée des macros syntaxiques fut introduite par S. SCHUMANN [20] et un processeur expérimental a été implémenté par P. CHATELIN et moi-même.

Dans les chapitres second et troisième un essai est tenté pour appliquer la technique de définition-via-traduction à PL/1. C'est en quelque sorte un pari. Il y a deux parties à définir : (1) un langage de base sémantiquement équivalent à PL/1 qui servira de cible à la traduction, et (2) un ensemble de macro définitions pour spécifier comment un texte PL/1 doit être réduit dans le langage de base équivalent. Le second chapitre est consacré à la description du langage de base et les raisons qui ont conduit au choix de ses propriétés. Le langage de base (appelé BL/1) pourrait exister à n'importe quel "niveau" entre PL/1 et le code machine. Le niveau choisi induit un compromis entre la complexité de la définition de BL/1 et la complexité de la traduction de définition. D. SERAIN a partiellement implémenté une version de BL/1 en projet de DEA [21].

Au troisième chapitre le problème de la définition de macros pour la traduction de PL/1 en langage de base est abordé. Il est montré par là quelques problèmes réels pour utiliser les macros-syntaxiques. Des exemples typiques seront examinés en détail afin d'obtenir quelque base pour évaluer la technique de traduction utilisée. L'ensemble des transformations utilisées sont résumées en annexe.

La dernière partie de cette thèse concerne plus particulièrement la confrontation de ces idées avec les problèmes soulevés par la production d'un compilateur Algol 68 lorsque je fus directement en relation avec M. SIMONET et P.Y. CUNIN puis plus tard avec J. VOIRON et M. DELAUNAY.

Au quatrième chapitre je discuterai l'usage des arbres d'opérateurs comme une forme de code intermédiaire pour Algol 68. Cette forme assez intéressante est en relation directe avec les langages préfixés (forme Polonaise) que j'utilise comme pas intermédiaire dans la traduction de PL/1 en BL/1. Beaucoup de mon expérience acquise en travaillant avec PL/1 fut exploitée en développant le compilateur Algol 68. Bien entendu le code intermédiaire d'Algol 68 doit être "commode" et un effort particulier fut passé à le segmenter afin que l'arbre d'opérateurs tout entier ne soit pas généré en une fois.

Algol 68 a développé la manipulation des conversions de données (modifications) plus loin et bien plus systématiquement que n'importe quel langage général connu jusqu'ici. Il en résulte que l'écrivain de compilateur doit faire face à des problèmes tout à fait nouveaux et uniques. Pour la version de départ d'Algol 68, j'ai écrit un ensemble de routines (en Algol 68) qui décrit comment les modifications de données doivent être déterminées par le compilateur. Cette description documente cette première version. Plus tard un autre ensemble de routines ont été écrites, en tout premier lieu par P.Y. CUNIN et M. SIMONET pour prendre en compte la version révisée d'Algol 68 [22]. Algol 68 est un bon langage de documentation, mais plutôt que d'inclure toutes ces routines dans cette thèse, j'ai utilisé des macros-syntaxiques au chapitre cinquième pour accomplir la même tâche. L'intention est ici de démontrer les possibilités de définition d'un macro système lorsque l'aspect de l'implémentation a moins d'importance (le compilateur contiendra assurément des routines écrites à la main qui suivront de plus près l'algorithme décrit en Algol 68).

Au sixième chapitre quelques remarques sont faites sur ce qu'il y a au-delà du niveau du code intermédiaire (ou dans l'implémentation du langage de base - ce qui revient au même). Les techniques de génération de code sont discutées à la lumière d'Algol 68, mais il deviendra rapidement évident que les mêmes méthodes devraient être appliquées pour l'implémentation de BL/1 ou de n'importe quel autre langage intermédiaire.

Au chapitre final les macros syntaxiques sont revues à la lumière des applications qu'on leur a trouvé. Une forme améliorée de macros-syntaxiques est proposée et quelques exemples montreront comment elles pourraient fonctionner.

0.2. La relation avec les langages extensibles

L'utilisation de macros pour définir et implémenter un langage est une méthode proche de celle des langages extensibles. En fait, au cours du premier travail que j'ai effectué avec S. SCHUMANN,

nous avions choisi le point de vue suivant lequel PL/1 pouvait être considéré comme un langage étendu d'une base hypothétique. Compiler PL/1 devenait alors découvrir les extensions et les réduire successivement dans leurs équivalents en langage de base. Pourtant un langage de base pour un système extensible contient un mécanisme d'extension, partie inhérente du langage. Les macros sont généralement considérées cependant comme "externes" au langage cible.

Le projet de P. JORRAND et S. SCHUMANN dans un contexte plus large de langages extensibles était de considérer alors que l'extension se divisait en deux types : sémantique et syntaxique [20]. JORRAND s'occupait lui-même de l'aspect sémantique et la récente thèse de D. BERT traite entre autres de ces mêmes sujets [2]. Les extensions syntaxiques de SCHUMANN conduisent directement au développement des macros-syntaxiques. Un autre travail dans cette direction est accompli avec P. COUSOT [5].

La raison de cette dichotomie en est que lorsqu'un programme en langage de base est exprimé avec ses aspects sémantiques correctement explicités, c'est-à-dire avec tous ses constructions de modes, ses définitions d'opérateurs et ses spécifications de conversion, il est évident qu'aucun programmeur ne voudra jamais écrire un tel programme. Ainsi les extensions syntaxiques sont-elles introduites comme un moyen d'"adoucir" la forme externe pour l'utilisateur.

Comme on le verra au chapitre BL/1, mon propre travail a son point de départ véritable dans cette balance entre extension syntaxique et sémantique. BL/1 est simple sémantiquement. Tout le travail est effectué pendant la traduction par les macros-syntaxiques. C'est une politique délibérée semblable au "test de rupture" sur certaines machineries pour observer le point où "ça casse". En tenant de faire faire au mécanisme syntaxique le plus gros du travail, une compréhension plus claire de sa nature est apparue.

1. MACROS-SYNTAXIQUES

Les macros syntaxiques étant la structure d'accueil de la technique de définition et d'implémentation, il est raisonnable de commencer à expliquer clairement ce qu'elles sont et comment elles se rattachent à d'autres méthodes de macrotraitement. Dans ce qui suit les notions de base sont établies en utilisant des exemples simples et abstraits issus de PL/1 pour rendre compte de l'approche fondamentale. Puis on montre comment l'usage de prédicats augmente notablement le pouvoir expressif des macros-syntaxiques. L'implémentation actuelle du processeur de macro-syntaxiques dépasse cependant le cadre de cette thèse. C'est essentiellement le travail accompli par P. CHATELIN [3]. Il sera néanmoins montré ici que l'usage d'extensions syntaxiques facilite et simplifie la manière d'implémenter le système. Ces méta-macros sont en elles-mêmes une démonstration de l'utilité des macros-syntaxiques.

1.1 Les Principes -

Une notion de base est qu'une règle syntaxique appartenant à une grammaire hors-contexte peut être considérée comme un appel de macro en plus de sa fonction analytique classique.

Considérons par exemple la règle :

facteur → primaire '↑2'

d'une grammaire d'expressions arithmétiques qui n'autorise aucune forme d'exponentiation. C'est aussi bien une macro qui est appelée pendant l'analyse au moment où la phrase

primaire ↑ 2

est reconnue. L'analyse appartenant au non terminal <primaire> est le paramètre effectif de l'appel de macro. Pour compléter tout à fait l'analogie avec les macros classiques il doit y avoir un "corps" de

macro qui remplace la phrase originelle par une nouvelle. On notera cette dernière à la droite de la règle après un signe égal "=".

Par exemple, le carré du <primaire> ci-dessus pourra être remplacé par le <primaire> au carré :

facteur → primaire '↑2' = primaire '*' primaire

Une fois le remplacement en place l'analyse syntaxique continue : elle reprend à partir de la tête de la nouvelle chaîne d'entrée. Ce qui suit cette dernière sera analysé après par conséquent, si aucun autre appel de macro intervient, alors doit apparaître une analyse commençant par le non-terminal <facteur> qui couvre au moins le début des remplacements. Pour l'exemple, ce serait l'application des règles (non-macro) :

facteur → facteur '*' primaire | primaire.

Lorsque l'entrée toute entière est reconnue en utilisant les règles sans remplacements, l'analyse est terminée. Ces règles qui apparaissent dans l'analyse finale sont les règles de base (la précédente dans cet exemple).

L'ensemble des règles de base décrivent le langage de base. Ce langage est en quelque sorte le langage cible produit par les appels de macros-syntaxiques pendant l'analyse de l'entrée. Un cas "dégénéré" est un texte entièrement écrit en langage de base et soumis au macro-analyseur. Il est analysé alors entièrement de manière classique.

Si à l'ensemble des règles de base, il est ajouté les règles macros alors cette grammaire plus les remplacements décriront le langage étendu.

Un texte écrit dans ce langage sera analysé comme s'il avait été écrit dans un langage qui est un sous-ensemble de lui-même : le langage de base. Les macros-syntaxiques servent à établir une correspondance (mapping) entre des expressions du langage étendu et leur équivalent en langage de base.

Il est à noter ici la différence entre cette traduction et celle d'un traducteur classique distinguant langage source et langage cible. Pour une macrogrammaire le texte source est analysé comme s'il était déjà en langage cible et le résultat est une analyse en son équivalent en langage de base.

Si le système en question possède une phase ultérieure nécessitant une analyse du langage cible, le macro-analyseur permet de sauter cette étape.

1.11 - Un simple exemple issu de PL/1 -

Ces principes seront mieux compris en se référençant à un exemple concret. Cet exemple est choisi pour illustrer aussi ce qu'il advient lorsque la nouvelle analyse (après l'appel de macro) ne couvre pas complètement la phrase de remplacement et pourquoi c'est une technique utile.

En général cette "réanalyse" peut être plus ou moins longue que celle de la phrase de remplacement. Plus courte : il est compréhensible que le reste sera analysé par un non-terminal de plus haut-niveau ; plus longue : alors la nouvelle analyse pour le non-terminal macro incluera une part de la chaîne d'entrée.

Considérons maintenant l'exemple PL/1.

Supposons un compilateur pour un sous-ensemble de PL/1. C'est un sous-ensemble parce qu'en déclarant les tableaux on demande de donner les bornes inférieures et supérieures pour chaque dimension. Donc la valeur "un" par défaut pour une borne inférieure manquante n'est pas permise.

Une partie de la grammaire du langage de base est :

liste bornes → bornes ',' liste bornes | bornes

bornes → expression ':' expression

expression → ...

Le but à atteindre est la modification de l'analyse de telle sorte qu'il ne soit plus nécessaire de spécifier la borne inférieure : un par défaut.

La modification peut être accomplie en ajoutant les deux macros-syntaxiques suivantes :

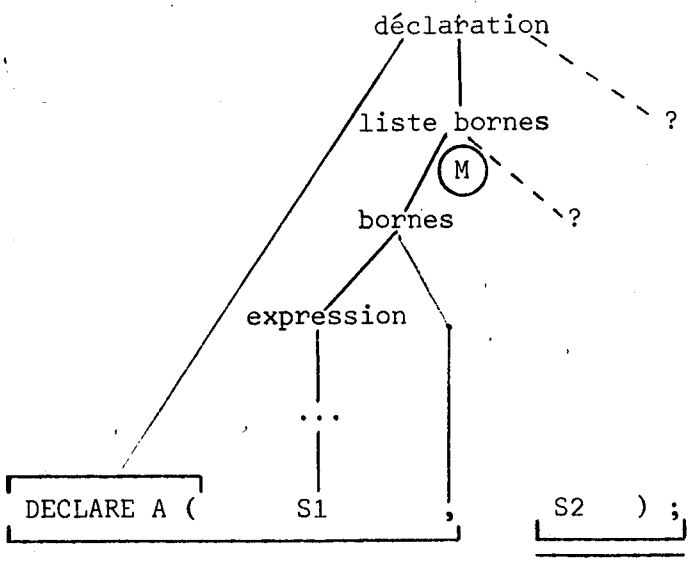
(1) bornes → expression ',' = '1:' expression ','

(2) bornes → expression ';' = '1:' expression ';'

Supposons qu'à la nouvelle macro-grammaire on soumette :

DECLARE A(S1, S2) ;

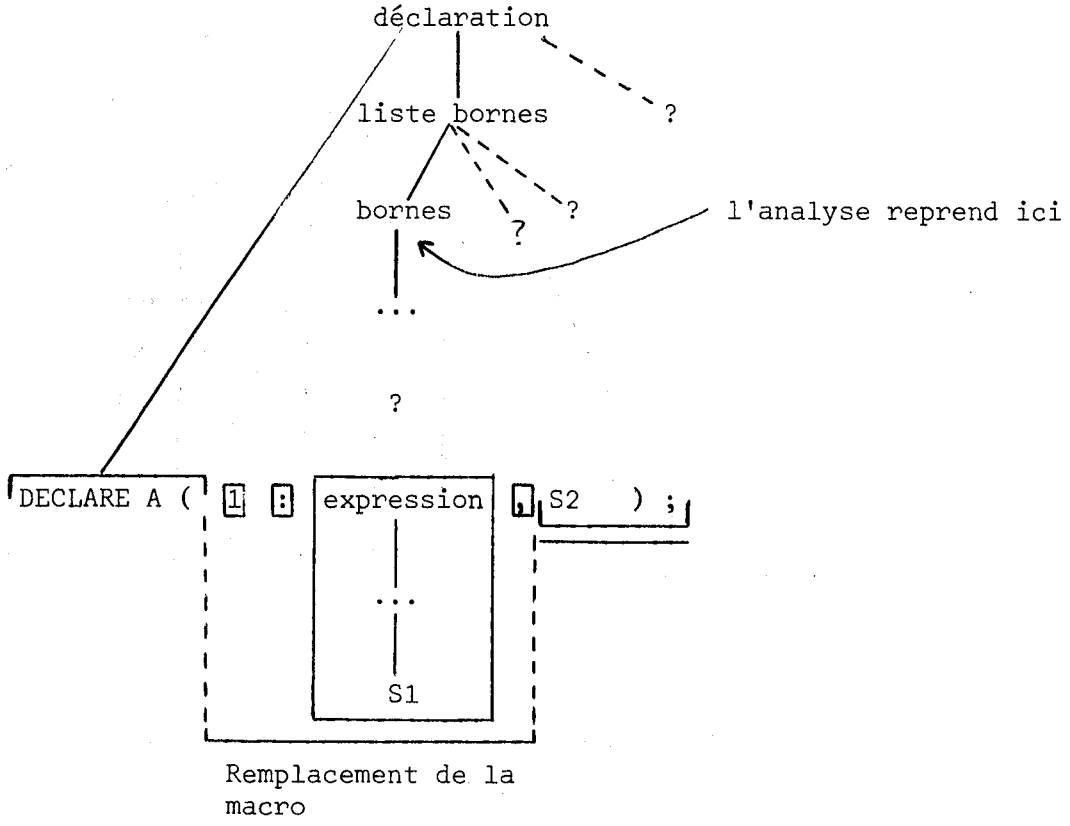
L'état de l'analyse au premier appel de macro sera (Ⓜ indique l'appel de macro) :



Partie analysée de l'entrée lorsque la macro (1) est appelée.

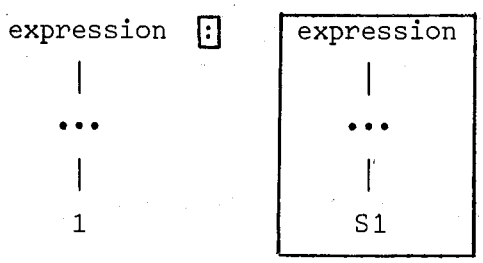
Partie non analysée.

Après l'application de la macro le résultat est :



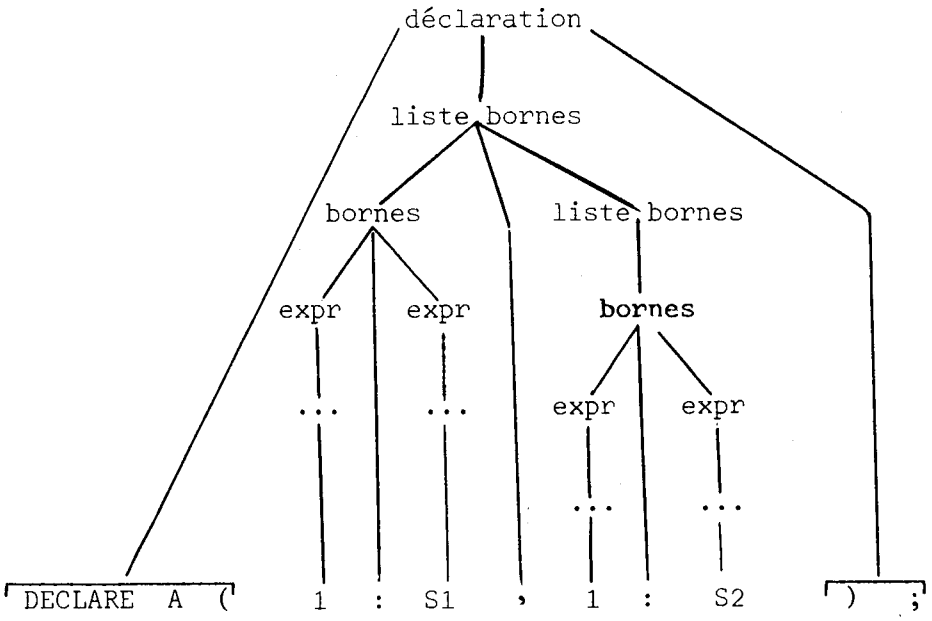
les paramètres de macro sont encadrés ([ ]).

La nouvelle analyse commençant par <bornes> reconnaît alors la phrase



dans la nouvelle chaîne d'entrée. Le dernier paramètre [,] sera inclu dans l'analyse de <liste bornes> située juste au dessus. L'analyse continue par un second appel de macro, cette fois-ci pour macro (2).

L'analyse se termine finalement avec :



Ce résultat est composé seulement de règles de base et par là peut être généré correctement en utilisant les macanismes déjà disponibles dans le compilateur.

Si, d'autre part, on avait ajouté simplement la nouvelle règle au compilateur :

bornes → expression

il aurait été nécessaire de modifier la portion de génération qui prend en compte la sémantique introduite pour la nouvelle construction. Cet exemple particulier est d'un intérêt limité mais il montre qu'effectivement une partie de la sémantique peut être traité via une conversion syntaxique.

1.2 Les remplacements conditionnels -

Si une règle de syntaxe est considérée comme une macro on peut continuer l'analogie plus loin.

En particulier, il est fructueux d'être capable d'écrire des macros seulement applicables lorsqu'un prédicat est vrai. Par exemple :

A → B 'C' D = si P alors B D sinon D B fsi

est un appel de macro.

- La phrase originale "B 'C' D" est reconnue pendant l'analyse.
- Le prédicat P (un nom de fonction) est évalué par la suite.
- Si le prédicat est vrai alors la phrase "B D" est utilisée comme remplacement.
- Sinon c'est la phrase "D B".

Comme on peut s'y attendre par la forme donnée au remplacement conditionnel il est possible d'emboîter de nouvelles conditions dans l'une et/ou l'autre clause alors et/ou sinon.

Par exemple :

```
A → B = si P1
        alors R1
        sinon si P2
              alors R2
              sinon R3
              fsi
        fsi
```

Cela signifie que :

R_1 est choisi lorsque P_1 est vrai

R_2 est choisi lorsque P_1 est faux et P_2 vrai

R_3 pour P_1 et P_2 faux

Comme en Algol 68, ceci peut conduire à un grand nombre de fsi à la fin de la clause pour un grand nombre d'emboîtements. On évite cela en combinant les mots-clefs sinon et si en sinsi et un fsi est éliminé en fin de clause (ce sera rigoureusement défini au paragraphe 1.5 sur la métasyntaxe).

L'exemple précédent s'écrit par conséquent :

$A \rightarrow B$ = si P_1 alors R_1 sinsi P_2 alors R_2 sinon R_3 fsi

La contraction alorsi est utilisée pour alors si

Lorsque le prédicat n'est pas satisfait (faux) il arrive souvent qu'aucun remplacement ne soit produit. L'analyseur devrait continuer comme si la phrase n'avait jamais été reconnue. Ceci est indiqué en écrivant échec au lieu et place d'un remplacement.

Un exemple de la construction utilisée sera :

si P alors R sinon échec fsi

qu'on abrègera en :

si P alors R fsi

On peut utiliser un prédicat seulement pour vérifier l'état d'une construction syntaxique sans produire de remplacement. L'analyse reprendra sur le texte comme si l'appel de macro n'existait pas. On indiquera sans pour spécifier la reprise sans remplacement :

si P alors sans fsi.

Notons qu'un remplacement peut être vide. Tout se passe comme si toute la chaîne d'entrée reconnue par le non-terminal macro était ignorée. La reprise de l'analyse s'effectue néanmoins à partir du non-terminal macro sur l'entrée :

si P alors vide fsi

1.3 Les prédicats -

A la conférence AFCET 70, S. SCHUMANN proposa huit types de prédicats pour les macros [15]. Dans l'implémentation actuelle nous avons constaté que deux seulement sont utilisés constamment. Ils seront décrits ci-dessous et on montrera sur les prédicats unifiés (1.6) qu'on peut les considérer comme deux aspects d'une même construction.

1.31 - Les fonctions-prédicats -

Cette première forme de prédicat est surtout un mécanisme d'appel pendant l'analyse d'une routine écrite à la main. La grammaire doit contenir une déclaration d'un (méta) identificateur qui est un nom de fonction.

Par exemple "fonpred" est défini comme un nom de fonction par l'instruction :

fonpred → * f9

"*f" est une dénotation pour la déclaration de la fonction de numéro 9 qui identifie la routine à appeler.

En plus de sa valeur booléenne une fonction-prédicat peut avoir des arguments syntaxiques et un résultat syntaxique. Par "syntaxique" on entendra spécification de non-terminaux qui ont, ou qui auront, comme valeur une analyse syntaxique. Une fonction prédicat typique est appelée en écrivant :

résultat : fonpred (arg1, arg2, ...)

Arg1, ... sont des noms de non-terminaux arguments de l'appel de la fonction-prédicat. Chacun de ces non-terminaux doit posséder une analyse et une valeur. Le non-terminal délivré comme résultat syntaxique de l'appel est appelé "résultat". Sa valeur (analytique) est établie par l'appel de la fonction-prédicat.

Des formes dégénérées de fonction-prédicats sont permises. Citons :

fonpred (arg1, arg2, ...)

résultat : fonpred

et plus simplement encore :

fonpred

peut aussi être donné lorsque les arguments et les résultats ne sont pas nécessaires.

L'usage des fonctions-prédicats peut être illustré par la fonction "concaténer". Cette fonction de deux arguments aux mêmes définitions syntaxiques :

liste → liste 'e' | 'e'

donne un résultat de même définition dont la valeur est l'analyse résultant de la concaténation des deux listes :

liste : concatene ($liste_1$, $liste_2$)

La macro-règle écrite en :

paquet → '(' $liste_1$ '+' $liste_2$ ')'

= si $liste_3$:, concatene ($liste_1$, $liste_2$)

alors '(' $liste_3$ ')'

fsi

signifiera qu'une chaîne d'entrée telle que :

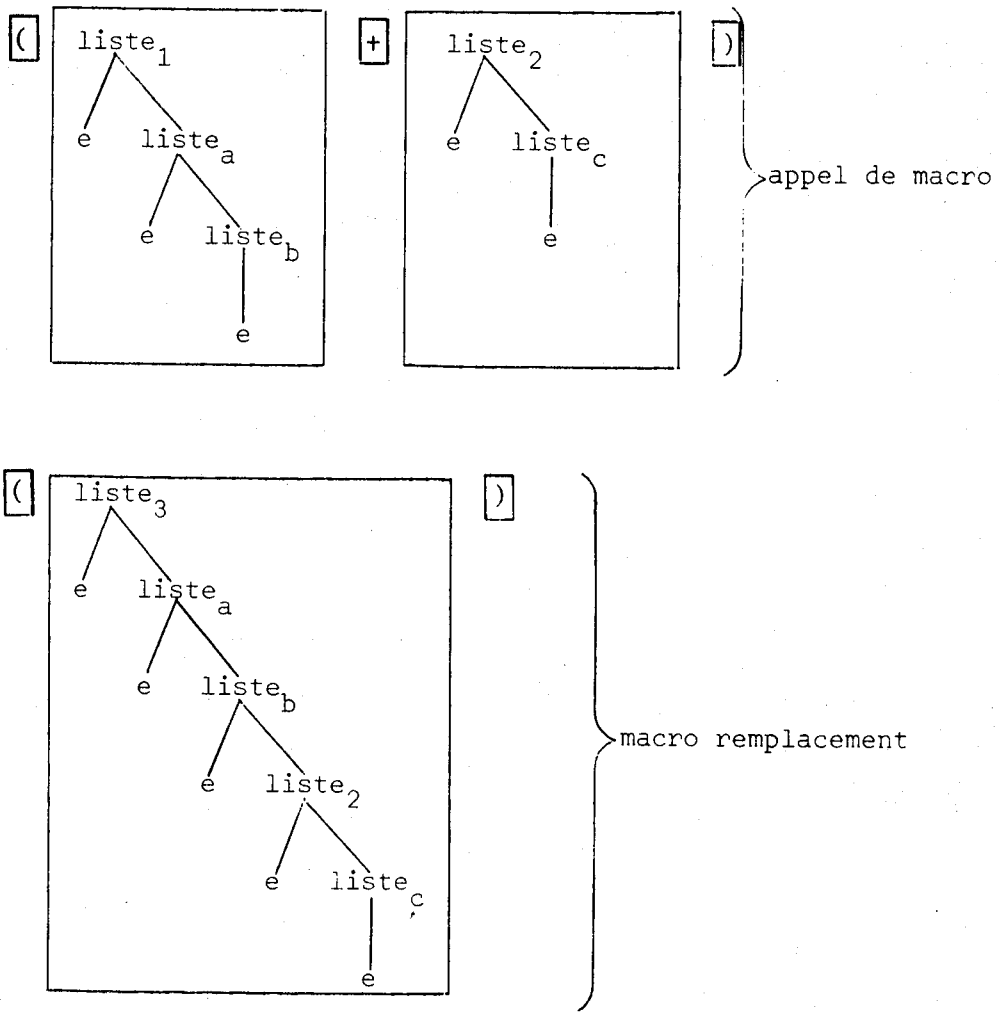
(e e e + e e)

sera traitée pour être reconnue en :

(e e e e e)

On notera que "concaténe" a toujours la valeur booléenne vraie.

En terme d'arbres syntaxiques la transformation s'opère ainsi :



Remarque : On peut également accomplir cette "concaténation" sans fonction-prédicat par application répétitive de macros. Mais ce n'est ni pratique ni efficace.

1.32 - Les prédicats de sous-structure -

Un non-terminal paramètre de macro peut être testé pour rechercher sa sous-structure particulière. La valeur booléenne du prédicat de sous-structure sera vrai ou faux suivant l'arbre construit.

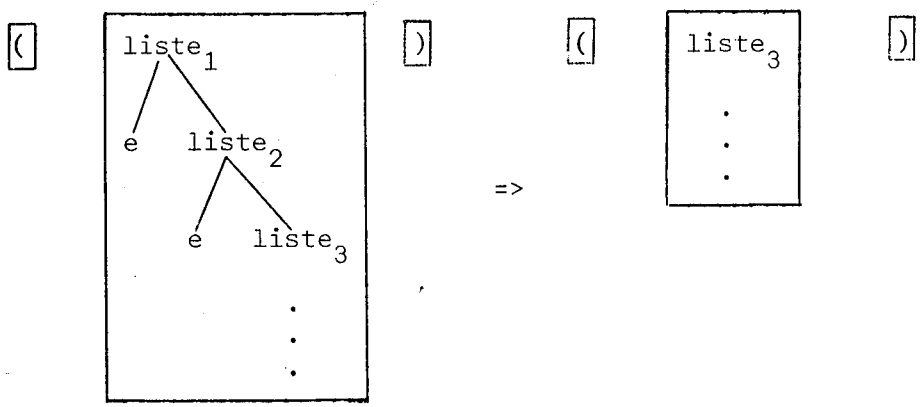
Ces prédicats ont un autre usage : créer les nouveaux noms de paramètres issus des sous-phrases du non-terminal en question. On peut appliquer à ces derniers de nouveaux prédicats de sous-structure pour déterminer les leurs etc....

Ainsi un mécanisme est-il disponible pour explorer tout sous-arbre syntaxique.

En voici un exemple :

paquet → '(' $liste_1$ ')'

= si $liste_1$ → 'e' $liste_2$
alorsi $liste_2$ → 'e' $liste_3$
alors '(' $liste_3$ ')'
fsi



(appel de macro) (macro remplacement)

Notons que les prédicats sont utilisés simplement pour donner un nom à l_3, afin d'obtenir la transformation désirée, c'est-à-dire éliminer deux éléments de la liste. Cette macro s'applique récursivement afin de réduire <paquet> à un ou deux éléments. Notons également que le mot-clef alorsi contraint l'évaluation du prédicat à sa gauche avant celle du prédicat à sa droite. C'est nécessaire parce que la valeur de $<list>_2$

doit être établie avant que la sous-structure puisse être déterminée par le second prédicat. L'opérateur booléen et (∧) ne pouvait être utilisé dans ce cas : aucune hypothèse n'étant faite quant à l'ordre d'évaluation des opérandes.

1.33 - Un exemple PL/1 utilisant des prédicats -

Supposons à nouveau que le compilateur n'implémente qu'un sous-ensemble de PL/1. Et dans ce cas supposons qu'il manque l'affectation multiple à gauche. Les macros-syntaxiques à écrire vont donner au compilateur cette affectation sans modification de sa partie sémantique. La part correspondante de la syntaxe de base du compilateur est :

```
listeinstr → instr ';' listeinstr | instr ';'
instr → affectation | ...
affectation → identificateur '=' expression
expression → ...
```

Les macros requises analysent l'affectation multiple comme une liste d'affectations simples. Pour accomplir ceci, deux règles "de transition" doivent être ajoutées à la syntaxe :

```
pg → identificateur ',' pg
pg → identificateur ',' identificateur
```

La distinction entre ces règles et les règles de base est qu'elles n'apparaissent jamais dans l'analyse finale du programme. Elles servent de support pour d'autres macros. Puisque ces règles doivent disparaître de l'analyse on les appelle aussi "macros sans remplacement" (d'une manière abusive).

La macro récursive réduisant la partie gauche peut maintenant être décrite :

affectation → pg_1 '=' expression ';'

= si pg_1 → identificateur ',' pg_2
alors identificateur '=' expression ';'
pg_2 '=' identificateur ';'
sinsi pg_1 → $identificateur_1$ ',' $identificateur_2$
alors $identificateur_1$ '=' expression ';'
$identificateur_2$ '=' $identificateur_1$ ';'
fsi

Les prédicats déterminent quelle alternative de <pg> appliquer et provoquent le choix du remplacement approprié. S'il y a au moins trois identificateurs en partie gauche alors la macro s'applique à nouveau.

Si la chaîne d'entrée est :

X, Y, Z = A + B ;

alors elle est analysée en :

X = A + B ; Y = X ; Z = Y ;

Remarque : en réalité les choses sont un peu plus compliquées à cause des conversions de données et des effets de bord. Ceci est ignoré dans l'exemple par souci de simplicité.

1.4 Liaison de Paramètres -

Puisque la notion de liaison de paramètre utilisée dans notre implémentation des macros-syntaxiques est plus évoluée que celles que l'on trouve généralement dans les appels de macro, elle mérite d'être décrite avec soin ici.

On distinguera dans le texte d'une macro syntaxique deux façons d'utiliser des identificateurs désignant un non-terminal :

(1) occurrences de définition.

(2) occurrences d'utilisation.

Une occurrence de définition d'un identificateur d'un non-terminal associe ce paramètre formel à un arbre syntaxique (en d'autres termes une valeur est établie). Les non-terminaux de la phrase originale (la partie droite de la règle H.C.) d'une macro règle sont tous des occurrences de définition.

Une occurrence d'utilisation d'un identificateur d'un non-terminal est une utilisation d'une valeur qui a été établie à partir d'une occurrence de définition quelque part. Les non-terminaux de la phrase de remplacement sont tous des occurrences d'utilisation.

C'est bien sûr essentiellement le même phénomène que l'on trouve avec les déclarations et les utilisations d'identificateurs dans les langages de programmation classiques quoique d'une manière légèrement différente. La situation est plus compliquée quant aux prédicats. Considérons les deux types de prédicats déjà discutés :

(1) $\underbrace{S_1}_{U} \rightarrow \underbrace{S_2 \quad S_3 \quad S_4 \quad \ldots}_{D}$

(2) $\underbrace{S'_1}_{D} : FP \ (\underbrace{S'_2, \ S'_3, \ S'_4, \ \ldots}_{U})$

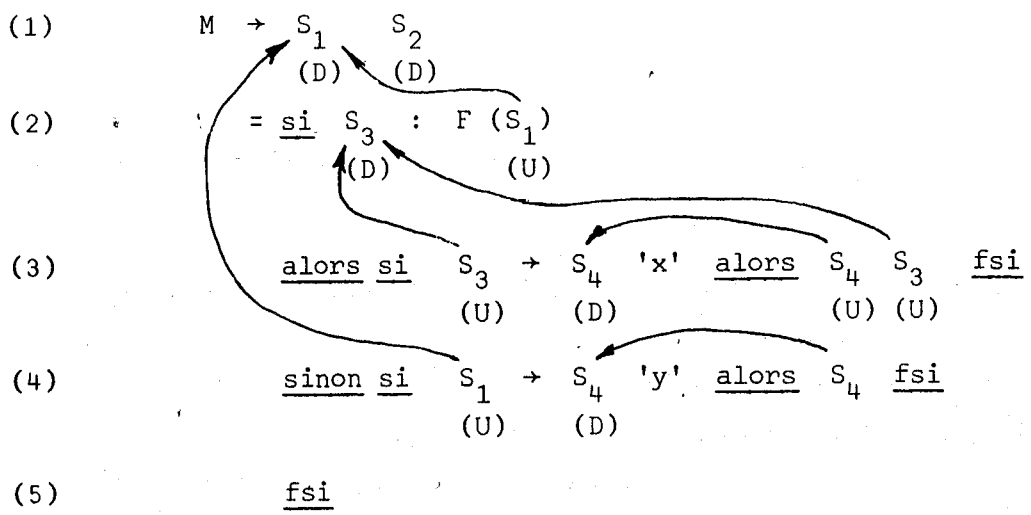
Nota : U et D pour occurrences d'Utilisation et de Définition. Pour le prédicat de sous-structure S_1 est une utilisation d'un paramètre dont la valeur a déjà été établie. S_2, S_3, ... reçoivent une valeur si le prédicat est vrai. Pour la fonction-prédicat c'est le résultat S'_1 qui est défini et qui reçoit une valeur si la fonction retourne vrai ; les arguments S'_2, S'_3 ... sont traités comme des occurrences d'utilisation, c'est-à-dire ce sont des paramètres d'entrée dont les valeurs ont déjà été établies - Ceci est une décision arbitraire utile pour accroître la sécurité du système et qui semble correspondre à la plupart des cas réels d'utilisation des fonctions prédicats.

C'est essentiellement la liaison dynamique qui a été discutée à ce point : un identificateur est associé à une valeur après qu'un ensemble de prédicats aient été évalués. Il y a cependant un aspect statique de la liaison qui est aussi intéressante. Il est possible d'avoir deux non-terminaux de même nom auquels sont associés deux valeurs différentes. Prenons par exemple une règle issue du dernier exemple :

pg → identificateur ',' identificateur

Lorsque cette règle a été utilisée pour un prédicat de sous-structure, des indices ont été ajoutés à chaque non-terminal "identificateur" afin de les distinguer correctement. Un paramètre impliqué dans toute occurrence est identifié par une paire : nom du non-terminal, indice.

La portée d'une paire n'est pas nécessairement le corps entier de la macro. Si le paramètre est défini par un prédicat alors il n'est pas besoin de l'appliquer hors de la clause si courante. (Notre implémentation restreint à la clause alors cette portée, j'en suis pourtant venu à penser que c'est peut-être trop rigide). Illustrons cela par la macro suivante où (U) et (D) représentent "utilisation" et "définition" et les flèches montrent l'association adéquate.



A noter qu'aux lignes (3) et (4) les S_4 sont deux exemples distincts de S parce que leurs occurrences de définition se trouvent dans des clauses si disjointes. Si deux paires de même nom sont utilisées dans des clauses imbriquées alors la plus interne est utilisée pour chaque association.

Tout ceci suit des pratiques bien établies dans les langages à structure de bloc, en particulier en Algol 68.

On a vu ici que ces mêmes procédures de définition sont utiles lorsqu'on les applique à une création inhabituelle comme celle du langage de description des macros-syntaxiques.

1.5 La méta-syntaxe -

Le langage utilisé pour spécifier les macros-syntaxiques, appelé le méta-langage en bref, est en lui-même un exemple éclairant des techniques d'implémentation et de définition, thème de cette thèse. Pour cette raison on le développera ici en détail.

1.51 - La grammaire de base -

Considérons d'abord la grammaire suivante qui définit un langage d'écriture de grammaires hors-contexte :

```
grammaire → listederègles ';'          *I 1 ;
listederègles → listederègles ';' règle      ;
listederègles → règle                        ;
règle → règlepropre '*I' nombre        *I 7 ;
règle → règlepropre                          ;
règle → nom '→' '*L' nombre            *I13 ;
règlepropre → nom '→' phrase           *I 8 ;
```

```
phrase → phrase unité                    ;
phrase → unité                           ;
unité → nom                        *I11  ;
unité → chaîne                     *I12  ;
nom   →  *L1                             ;
nombre → *L2                             ;
chaîne → *L3                             ;
```

Le langage dans lequel cette grammaire est écrite est le même que celui que définit la grammaire.

Les trois dernières lignes de la grammaire déclarent "nom", "nombre", et "chaîne" comme pseudo-terminaux (c'est-à-dire reconnaisseurs lexicographiques). Cela signifie que les nombres donnés après "*L" sont des codes internes des routines utilisées pendant l'analyse. Ces routines reconnaissent chaque unité lexicographique. Elles ont les définitions informelles suivantes :

nom - une lettre suivie par une suite de lettres ou chiffres (et seulement celles-ci ou ceux-là)

nombre - une suite de chiffres (et seulement des chiffres)

chaîne - une marque unique (') suivie par un nombre quelconque de caractères sauf la marque (') qui la termine (la marque elle-même est spécifiée seule par '''', et par '' dans la chaîne).

Quelques-unes des règles de la grammaire ci-dessus se terminent par la notation "*I" suivie d'un entier. Lorsque l'analyse est terminée et le langage de base équivalent trouvé, ces nombres définissent les fonctions sémantiques à appeler pour effectuer la génération (synthèse) correspondant au programme en langage de base. Ces nombres sont appelés nombres de synthèse. Chacun correspond à une fonction de synthèse. L'ensemble de ces fonctions pour un langage de base donné est appelé synthétiseur.

Une structure de contrôle standard est définie pour celui-ci. La synthèse commence par un appel à une fonction de synthèse attachée à l'axiome de la grammaire (de base). La partie droite de toute règle est une chaîne de non-terminaux et de terminaux. La fonction de synthèse pour cette règle de base peut appeler celle qui est attachée à la $n^{ème}$ unité. Ce doit être un non-terminal. Ce nombre de synthèse (de la $n^{ème}$ unité) dépendra de l'alternative où le non-terminal a été trouvé.

Un exemple fera mieux comprendre :

```
N → S1  S2  S3      *I13 ;

S2 → SX  SY  SZ     *I 7 ;

S2 → SA  SB         *I12 ;
```

La fonction de synthèse n° 13 peut contenir un appel à la routine attachée à la deuxième unité de la partie droite (le non-terminal <S2>). Ce sera la fonction de synthèse n° 7 si <S2> est la phrase

```
SX  SY  S2
```

sinon la fonction n° 12 sera appelée.

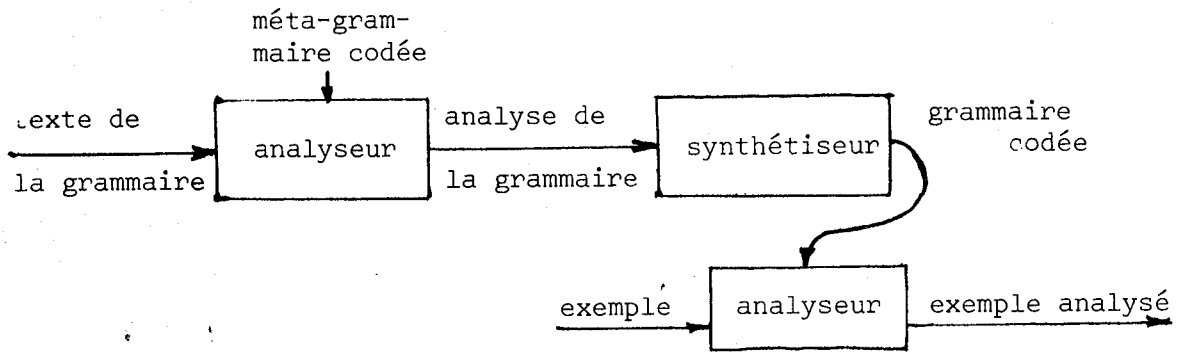
Une telle structure de contrôle peut être utilisée pour "cheminer" dans l'arbre d'analyse et produire toute génération désirée. Pour rendre ce processus très simple, une fonction de synthèse "blanche" a été introduite dans le système. Lorsqu'aucun nombre de synthèse n'est spécifié pour la règle de base, alors la fonction "blanche" de numéro "0" est présupposée. Cette fonction est simplement définie comme une suite d'appels des fonctions de synthèse des sous-non-terminaux de gauche à droite. Par exemple la seconde règle de la méta-grammaire ci-dessus aurait put être décrite :

```
listederègles → listederègles ';' règle *I0 ;
```

Ceci signifie que la première fonction de synthèse sera appelée pour <listederègles> et ensuite la fonction pour <règle>. Ce premier appel est potentiellement récursif (pour un niveau inférieur de <listederègle>). Cet exemple particulier est nettement un mécanisme assurant que les fonctions de synthèse de chaque règle du texte d'entrée sont appelés dans l'ordre gauche à droite. D'après leur nature particulière c'est soit n° 7, n° 13 ou n° 0 à nouveau.

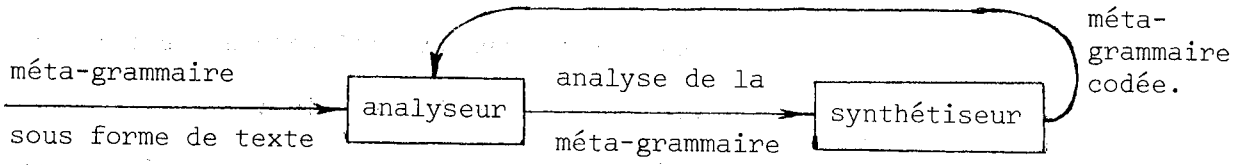
Comme mentionné plus haut, les fonctions de synthèse servent à n'importe quel propos dans la manipulation du texte analysé. Dans le cas de la méta-grammaire de base elles sont utilisées pour coder les règles de la syntaxe d'entrée en code numérique interne. Ce dernier forme les tables dont se sert le (macro) analyseur.

Le processus de préparation des grammaires codées et d'utilisation de celles-ci pour analyser un texte est schématisé ainsi :



La grammaire au début de ce processus est l'une des méta-grammaires possibles. Si cette méta-grammaire est analysée elle-même puis synthétisée, le résultat peut être utilisé comme méta-grammaire codée pour l'analyseur.

Schématiquement on obtient :



La fermeture de cette boucle n'est pas l'objet de la démonstration bien entendu. Afin d'initialiser l'usage de l'analyseur il est pourtant nécessaire de commencer par une version codée à la main de la méta-grammaire. Le gain réel provient des deux observations suivantes :

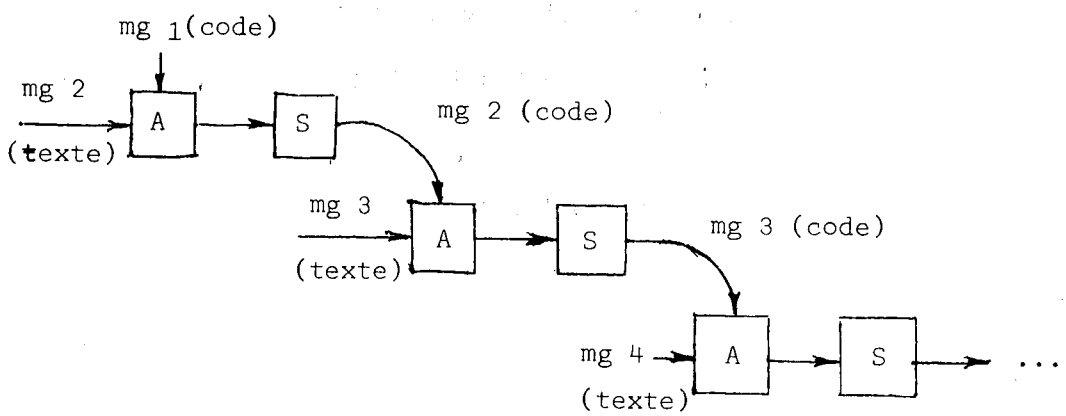
(1) La méta-grammaire de base peut être utilisée pour coder une méta-grammaire de loin plus complète et qui prendra en compte les règles macro, les prédicats, la règle epsilon ("règle vide") et ainsi de suite.

(2) Des macros-syntaxiques peuvent aussi être écrites pour la méta-grammaire. Elles permettent alors des extensions syntaxiques pour la description des grammaires et des macros syntaxiques elles-mêmes.

Chacun de ses aspects est décrit dans ce qui suit.

1.52 - Génération télescopique -

L'addition de mécanismes obtenus par extension de la méta-grammaire de base est un des avantages classiques des langages de systèmes écrits dans leur propre langage. Comme on peut s'y attendre la méta-grammaire peut être augmentée plus facilement par étapes successives :



Comme exemple prenons l'addition à la deuxième étape de l'expression des macros-règles (sans prédicat)

règle → règlepropre '=' remplacement *I5 ;

remplacement → phrase *I14 ;

à la méta-grammaire précédente. Une fois la nouvelle méta-grammaire analysée et synthétisée en utilisant l'ancienne méta-grammaire codée, on dispose d'un analyseur des macros-règles simples. (Apparemment, le non-terminal <remplacement> est inutile. Il est inclu pour permettre d'autres alternatives). La seule chose à programmer est l'écriture des sous-programmes correspondant aux nombres de synthèse 5 et 14. La structure de contrôle qui utilise ces primitives a été codée automatiquement en utilisant l'ancienne méta-grammaire.

Dans l'état actuel de l'implémentation les additions aux grammaires ont été accomplies en quatre étapes (voir appendice B). Il y a un seul ensemble de sous-programmes écrit en PL/1 pour la partie synthèse dont se servent toutes les versions des méta-grammaires. (pour les plus simples versions certains ne servent pas).

1.53 - Méta-macros -

Une fois le remplacement simple disponible dans le méta-langage, il devient possible d'écrire des macros qui étendent le méta-langage sans ajouter quoique ce soit au synthétiseur (on se souviendra toujours ici que la synthèse des règles syntaxiques correpond à la partie génération d'un compilateur classique).

Un exemple comme l'abréviation de :

$N \rightarrow \alpha \ ; \ N \rightarrow \beta \ ; \ N \rightarrow \gamma \ ;$

en

$N \rightarrow \alpha \mid \beta \mid \gamma \ ;$

est considéré comme une extension du méta-langage effectuée par la macro :

règle → nom '→' phrase '|'

= nom '→' phrase ';' nom '→'

Lorsque ce texte est codé en méta-grammaire alors cette grammaire peut être utilisée pour analyser les règles syntaxiques en utilisant la notation barre (|).

Revenons maintenant à la définition formelle de quelques constructions introduites informellement plus haut. Le méta-langage de base décrit les remplacements conditionnels suivant la méta-syntaxe :

remplacement → 'si' prédicat clausealors clausesinon 'fsi' *I16 ;

remplacement → 'échec' *I15 ;

clausealors → 'alors' remplacement ;

clausesinon → 'sinon' remplacement ;

Pour écrire :

si P alors R fsi

comme abréviation de :

si P alors R sinon échec fsi

on définit (et on implémente) la macro :

clausealors → 'alors' remplacement 'fsi'

= 'alors' remplacement 'sinon échec fsi' ;

De la même manière alorsi, sinsi peuvent être définis par :

clausealors → 'alorsi' prédicat clausealors clausesinon 'fsi'

= 'alors si' prédicat clausealors clausesinon 'fsi fsi' ;

clausesinon → 'sinsi' prédicat clausealors clausesinon 'fsi'

= 'sinon si' prédicat clausealors clausesinon 'fsi fsi'

Prenons l'exemple :

```
paquet → '(' liste1 ')'
       = si liste1 → 'e' liste2
         alorsi liste2 → 'e' → liste3
         alors '(' liste3 ')'
         fsi
```

présenté à la méta-grammaire augmenté des dernières macros, analysera en :

```
paquet → '(' liste1 ')'
       = si liste1 → 'e' liste2
         alors si liste2 → 'e' liste3
                alors '(' liste3 ')'
                sinon échec
                fsi
         sinon échec
         fsi
```

Ici à nouveau il devrait être clair qu'une notation utile a été ajoutée au méta-langage sans modification de la partie programmée du système.

1.54 - Adaptabilité -

En réalité un grand nombre de macros différentes ont été introduites pour tenter de résoudre par une notation ou une autre, quelque problème d'écriture de grammaires. Une, assez ingénieuse, écrite par exemple par P. CHATELIN applique des prédicats à des portions de phrases de remplacement et concatène les résultats entre eux.

Puisque certaines de ces macros satisfont certains et pas les autres, il n'y a pas en fait de version standard d'extension du méta-langage. Chaque utilisateur du système semble avoir sa propre version. La version que je propose en appendice est une des plus conservatives. On notera aussi que si une grammaire particulière n'utilise pas un aspect du méta-langage, la grammaire peut être codée plus rapidement en utilisant une version particulière (ou réduite) du méta-langage. Ceci peut être obtenu facilement par une technique de génération télescopique.

On peut prévoir qu'une prolifération de dialectes du méta-langage rendrait le système inutilisable. En fait, ce n'est pas arrivé et il semble que ce soit dû au mécanisme d'extension d'un même méta-langage de base. Peu importe la forme externe, le même module de synthèse est utilisé et le code interne a toujours le même format. Il est assez facile d'accompagner toute grammaire par la méta-grammaire qui décrit la manière de l'écrire si elle a une particularité quelconque. Notre expérience semble renforcer l'opinion des enthousiastes d'un langage extensible pour l'un de ses avantages : la fabrication sur mesure du langage au projet tout en conservant la compatibilité au niveau du langage de base.

1.6 - Prédicats unifiés -

En plus de leur commodité opérationnelle les extensions syntaxiques ont un effet plus profond sur notre travail en changeant notre attitude vis à vis des langages de programmation. C'est peut-

être le point le plus délicat à expliquer. En substance ce qui est arrivé c'est de développer beaucoup de constructions syntaxiques aux formes externes variées comme des manifestations différentes de la même "construction de base". La syntaxe abstraite du rapport de Vienne [25] participe de cet esprit. Pour concrétiser notre conception changeante des macros, les prédicats seront utilisés comme exemple.

La proposition originelle de SCHUMANN [20] suggère une grande variété de types de prédicats. En fait deux parmi ceux-ci se trouvent être les plus utiles (fonctions et détecteurs de sous-structure). Leur codage sont implémentés par des routines de synthèse séparées comme il est décrit au paragraphe 1.3. L'une de ces formes, la fonction-prédicat, est "ouverte" puisque toute action raisonnable pratiquement peut être programmée pour eux. De nombreuses fonctions ont été développées en fonction des besoins et de l'ingéniosité. Il y a une fonction par exemple qui imprime des messages à l'utilisateur lorsque le prédicat est appliqué (utile à la mise au point). Un autre teste l'égalité de deux arbres syntaxiques avec l'appel suivant :

égalité (N_1, N_2)

On est tenté à ce point d'exprimer cela d'une manière plus habituelle par :

$N_1 = N_2$

Il est assez simple de l'implémenter par la méta-macro :

prédicat → $unité_1$ '=' $unité_2$
= 'égalité(' $unité_1$ ',' $unité_2$ ')'

D'autres notations dans ce style furent rapidement trouvées pour d'autres fonctions-prédicats. Ceci soulève inévitablement la question : si la fonction prédicat peut être considérée comme la forme de base de nouveaux prédicats pourquoi ne pas l'utiliser pour tous les prédicats y compris ceux qui ont été spécialement implémentés au début ?

On peut le faire mais il y a ici une légère contrainte dans la forme de cette fonction prédicat. Comme on l'a vu, on écrit :

$$\underbrace{\text{résultat}}_{\text{occurrence de définition}} : \text{nomdefonction } \underbrace{(\text{arg}_1, \text{arg}_2, \ldots)}_{\text{occurrence(s) d'utilisation}}$$

Un seul résultat est permis et par conséquent un seul paramètre défini par cet appel. En fait, il est assez facile de généraliser cette forme pour retourner plusieurs résultats :

$$\underbrace{(\text{résultat}_1, \text{résultat}_2, \ldots)}_{\text{définition}} : \text{nom de fonction } \underbrace{(\text{arg}_1, \ldots)}_{\text{utilisation}}$$

On l'appelera forme unifiée de prédicat. Elle peut servir comme construction de base pour tout prédicat utilisé dans notre système. Par exemple un prédicat de sous-structure écrit en :

$$N \rightarrow S_1 \quad S_2 \quad S_3$$

peut être interprété par les méta-macros en :

$$(S_1, S_2, S_3) : \text{sous-structure } (N)$$

ce qui est intéressant ici, c'est que le codage résultant du prédicat unifié n'est pas différent en substance de celui que l'on obtiendrait en traitant chaque prédicat comme un cas particulier du module de synthèse. Un résultat en est une version plus courte et plus claire des routines de synthèse.

En conclusion ceci montre que l'usage d'extensions syntaxiques explicites peuvent avoir une influence utile sur le projet de programmation qui à son tour conduit à un software plus net.

1.7 - Transformations syntaxiques -

L'une des difficultés rencontrées en travaillant à l'aide de macros-syntaxiques est de voir ce qui est accompli à chaque appel et à chaque remplacement. Aussi comme aide purement notationnelle la convention de représenter les transformations par la chaîne de caractères (feuilles de l'arbre) sera souvent utilisé dans ce qui suit. Quelques détails syntaxiques sont perdus par conséquent mais le résultat est usuellement bien plus lisible. Cette représentation sera appelée "transformation syntaxique" ou en bref "transformation".

Comme exemple prenons la macro <paquet> décrite plus haut :

paquet → '(' l_1 ')'
= si l_1 → 'e' l_2
alorsi l_2 → 'e' l_3
alors '(' l_3 ')'
fsi

Une transformation effectuée par cet appel serait :

(e e e e) ⇒ (e e)

La double flèche (⇒) sépare le texte avant et après la transformation. On notera quelquefois des non-terminaux dans les chaînes lorsque leur nature est évidente ou sujette à conservation dans la transformation (notation minuscule ou majuscule suivant les cas). Reprenant cet exemple, <paquet> devient :

(e e l) ⇒ (l)

Ceci rappelle les grammaires de Chomsky type 1. La similarité repose sur ce détail (essentiel) qui a été supprimé dans la transformation : la partie gauche de l'appel de macro.

Ce n'est que lorsque ce non-terminal est prédit que le remplacement peut être effectué. Tandis que pour un simple mécanisme de réécriture de règle, toute phrase peut être remplacée dans le développement de la forme phrase. Nous ne pouvons étudier plus loin ici s'il existe une inclusion de l'un ou l'autre des langages (contexte liés - versus - macro-syntaxiques).

1.8 - Ambiguité faible -

Les grammaires de macros-syntaxiques, commes les grammaires hors contexte, peuvent conduire à des analyses ambigües d'un texte sauf si elles sont délibérément évitées dans la définition de la grammaire.

A la fin de la (macro) analyse il peut exister également différents arbres syntaxiques. On constate parfois qu'ils sont identiques. Que s'est-il passé ?

Considérons la macro suivante :

instr → idf_{pg} ':=' expr

= si idf_t : nouveau

alors idf_{pg} ':=' idf_t ':=' expr

fsi

(La fonction "nouveau" créée un identificateur nouveau et unique).

Appliquée au texte :

a := b + c ;

le résultat est clairement :

a := t := b + c ;

on atteint cependant ce résultat par deux transformations différentes :

soit : a := b ⇒ a := t := b

soit : a := b + c ⇒ a := t := b + c

Ceci parce que <expr> qui est le paramètre de la macro peut identifier soit "b" soit "b + c".

Lorsque les analyses finales sont identiques on les appellera "ambiguités faibles".

Ce cas peut arriver souvent malheureusement et passer inaperçu jusqu'au moment du test de la macro-grammaire. Il y a au moins deux solutions à ce problème :

(1) Ne jamais laisser un non-terminal à la fin de la règle macro sujet à des applications récursives (c'est-à-dire "a" est une <expr>, "a + b" est une <expr> etc...). On peut réécrire alors l'exemple précédent en prenant en compte le délimiteur :

$$\text{instr} \rightarrow \text{idf}_{pg}\ \text{':='}\ \text{expr}\ \text{';'}$$

$$= \underline{\text{si}}\ \text{idf}_{t}\ :\ \text{nouveau}$$

$$\underline{\text{alors}}\ \text{idf}_{pg}\ \text{':='}\ \text{idf}_{t}\ \text{':='}\ \text{expr}\ \text{';'}$$

$$\underline{\text{fsi}}$$

Le point virgule (;) n'autorise qu'une forme de <expr> à utiliser en paramètre. Ce n'est, hélas, pas toujours évident de connaître l'ensemble des délimiteurs possibles (considérer les expressions en ALGOL 68 !).

(2) D'ignorer simplement la deuxième ambiguïté !
Il est relativement facile de détecter l'ambiguïté faible de l'analyseur lorsqu'elle intervient et de la rejeter. Cette méthode sera implicitement utilisée lorsque les modifications d'Algol 68 seront discutées (Ch. 5).

2. UN LANGAGE DE BASE POUR PL/1

Les propositions pour les définitions et implémentations présentées jusqu'ici méritent un test sérieux. Pour cela un langage de programmation est nécessaire. Plutôt que d'inventer encore un autre langage "jouet" dont le seul projet serait fait "à la main" il a été décidé d'essayer PL/1. Ceci a certains inconvénients dont la taille n'en est pas le moindre. Mais d'autre part le "réalisme" n'est-il pas le but poursuivi ?

L'hypothèse de base est que PL/1 peut être considéré comme une extension syntaxique de quelque langage de base. Le processus de compilation devient l'usage d'un macro-analyseur pour "découvrir" ces extensions et produire le texte en langage de base qui lui est équivalent. Ce texte peut être compilé par un processeur beaucoup plus simple qu'un compilateur PL/1 avec "toutes ses plumes". Le problème essentiel est ici de décider du niveau où situer le langage de base. Il devrait être clair que n'importe quelle construction du langage peut être remplacée par un ensemble qui lui est sémantiquement équivalent mais formé de constructions plus simples (ces deux critères sont toutefois assez subjectifs). En écrivant plus de macros (définitions des équivalences sémantiques) on pourrait vraisemblablement aller à des niveaux de plus en plus bas du langage. Où est la limite ? On peut continuer théoriquement jusqu'au seul code numérique de la machine, assumant effectivement toute la compilation Il était désirable d'aller aussi loin que possible afin d'obtenir le test "fatal" mentionné en introduction. Mais, si d'autre part, le langage de base était encore indépendant de la machine cela aurait signifié que le travail effectué aurait été applicable à n'importe quel système (machine).

Nota : Il est intéressant de rapprocher cette observation de celle du langage du système de macro-syntaxiques décrite au chapitre précédent. Au lieu de "stopper" au niveau du méta-langage de base, il est parfaitement possible d'écrire plus de méta-macros encore qui traduiraient les règles de grammaire en forme numérique interne (exigée par l'analyseur). Ainsi tout ce qu'il resterait à la synthèse serait un "effeuillage" de l'arbre d'analyse à une chaîne de nombres.

Le langage de base présenté dans ce chapitre (BL/1 pour d'évidentes raisons) a été développé en fonction de certains critères :

- couvrir tous les concepts primitifs inhérents à PL/1,
- exprimer ces concepts d'une manière simple, à la fois syntaxiquement et sémantiquement,
- permettre une optimisation aisée si besoin est,
- permettre une génération rapide et en un passage (suivant les cas),
- être aussi indépendant que possible d'une machine particulière,
- être aussi indépendant que possible d'un système d'exploitation particulier,
- représenter d'une manière égale les différences mineures de dialectes PL/1.

Le système proposé a deux niveaux : l'un indépendant de la machine, l'autre dépendant. Le premier niveau, celui des macro-expansions, est complet par définition et peut être utilisé dans toute sa généralité. Le second niveau, l'implémentation de BL/1, doit prendre en compte l'environnement particulier du système et doit être fabriqué par des méthodes probablement classiques. Il est intéressant de noter que seule la portion générative d'une telle implémentation a besoin d'être prise en considération, puisque le processeur de macros-syntaxiques délivre un texte BL/1 déjà analysé*.

Il est concevable qu'on puisse désirer plusieurs implémentations de BL/1 pour une même machine.

* Ceci est le principe utilisé par D. SERAIN pour son implémentation partielle de BL/1 [21].

Une version par exemple servant à produire du "code" très vite avec de nombreux outils de mise au point pour vérification de programmes ; une autre pour produire du code "optimisé" pour les programmes d'un usage intensif.

L'optimisation est possible à deux niveaux. Comme l'implémentation spéciale de BL/1 l'a déjà suggéré, il est aussi possible d'imaginer un nouveau pas dans le système de compilation qui prenne la sortie en BL/1 du macro-processeur et la réorganise en un programme BL/1 plus efficace avant de soumettre au générateur de code*. A noter que les deux méthodes d'optimisation mentionnées ci-dessus correspondent très nettement à l'optimisation "machine-indépendante" et "machine-dépendante".

Bien des commentaires faits ici sont des réminiscences indubitables d'observations issues de systèmes de compilateurs de compilateurs. En effet pour un compilateur de compilateurs le "code objet" (produit par le compilateur résultant) n'est généralement pas vu comme un langage. D'autre part, le traitement par macros-syntaxiques se fonde sur des opérations linguistiques plutôt que sur une programmation explicite.

Un compilateur de compilateurs implique un prétraitement de la définition du compilateur afin de produire un processeur efficace. Bien que cela n'ait pas été envisagé dans le travail présent, ceci serait utile aussi dans le cas des macros-syntaxiques. Ceci revient à dire qu'on peut prendre un ensemble particulier de macro-définitions et les "geler" en un processeur particulier qui fonctionnerait notablement plus vite que l'analyseur général guidé par une table d'interprétation dont on dispose actuellement. Ceci pourrait être un intéressant sujet de recherche.

* C'est l'orientation de quelques travaux sur l'optimisation en cours avec M. EXEL.

2.01 - Origines de BL/1 -

La plupart des résultats présentés dans le reste de ce chapitre sont issus d'une recherche exécutée dans le cadre d'un contrat D.G.R.S.T. en 1969 et 1970. A cette époque, j'ai défini une version complète de BL/1 [ ]. Le langage est malgré tout assez long à présenter, aussi seules les parties importantes seront décrites ici. En particulier les on-conditions, le multi-tasking et les entrées-sorties sont omis car leur traitement est désormais classique.

L'une des conséquences inattendues de mon travail sur BL/1 est que j'en suis venu à noter de nombreuses ressemblances avec divers langages orientés vers la machine (et si différents) comme PL 360 [31], G S L (créé par le Centre Scientifique I.B.M. de Grenoble [ ]), les "machines abstraites" de P. POOLE. BL/1 a cependant été façonné indépendamment de ces derniers. Que BL/1 doive ressembler à PL/1 va de soi. Qu'il doive ressembler à la tendance actuelle ... est moins évident. J'ai tendance à regarder cela comme un signe encourageant qu'un langage intermédiaire général n'est pas aussi inaccessible que certains peuvent penser.

L'autre problème à mentionner ici est que BL/1 est déjà à revoir. Cela signifie que j'en ai un autre point de vue maintenant de part l'expérience acquise depuis. Je choisirais une autre approche quoique ce qui est décrit ici est encore satisfaisant pour le coeur du langage. Pour cette raison, je décrirai sa version actuelle avec de temps à autres quelques commentaires.

2.1 - Le prélude standard -

BL/1 emprunte au rapport Algol 68 la notion de "prélude standard". C'est-à-dire qu'un texte contenant certaines déclarations et opérations standard est préfixe par hypothèse à tout texte BL/1 particulier à compiler. Il y a une variété de raisons à cela.

(1) Des valeurs dépendantes de la machine peuvent être introduites. La version du compilateur PL/1 pour IBM 360 par exemple présuppose une

précision par défaut de (15,0) pour une variable FIXED BINARY. Ce serait certainement différent pour d'autres machines puisque ceci est lié à la taille du mot. Pour les précisions par défaut, le prélude standard doit contenir des modèles de déclaration auxquelles il faut se référencer en établissant les valeurs par défaut.

(2) Quelques structures de données peuvent être mieux organisées lorsque la machine est prise en compte. Les bits de conditions "enable/disable" sont représentées en BL/1 comme une structure (groupe). L'ordre de ces bits le plus convenable pour le 360 IBM serait celui du PSW. Ainsi une déclaration de structure peut être ajoutée au prélude standard pour refléter cette information.

(3) La communication standardisée entre des segments de programmes peut être établie. Si on désirait par exemple s'assurer que l'adresse de retour à la même adresse physique (un registre central, un emplacement mémoire) pour l'appel et la routine appelée ceci pourrait être effectué grâce à un nom standard déclaré pour cela dans le prélude.

A noter que si on suppose que le prélude standard est un texte BL/1 ceci est inexact puisque beaucoup de définitions en BL/1 sont à proprement parler hors langage. C'est la même approche utilisée dans le rapport Algol 68 lorsque par exemple, les sémaphores de Dijkstra sont définis seulement par un commentaire indiquant ce qu'ils font !

2.2 - Les descriptions de données -

En accord avec la philosophie générale de BL/1 le maximum a été fait pour conserver à la manipulation de données simplicité et efficacité. Aussi tous les types de données sont-ils construits à partir de dix formes primitives et toute opération sur ces constructions sont-elles faites explicitement avec peu d'hypothèses cachées.

La notion de base à traiter est celle de "descripteur". C'est un objet connu seulement par le processeur de base et non par le programme généré. Le descripteur donne l'information détaillée de l'objet (donnée) qui apparaît effectivement à la génération. Très généralement il contient :

- l'information pour générer l'adresse utilisée par le code
- la forme (attributs) de l'objet
- l'information utile pour l'optimisation du code
- des pointeurs pour les descripteurs en arbre.

Aucune de ces informations n'est laissée dans une forme qui serait préjudiciable à l'implémentation sur une machine particulière.

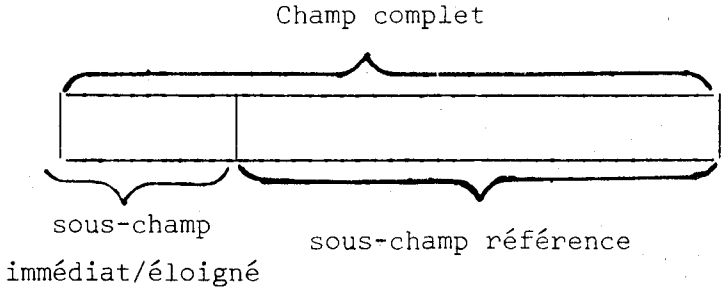
La ressemblance entre cette notion et celle d'une entrée dans une table de symboles n'est pas fortuite. Il est attendu de l'implémentation du processeur de base un mécanisme très voisin. La différence essentielle en est que BL/1 donne des primitives qui opèrent explicitement sur ces descripteurs pour créer de nouvelles définitions de données ou pour en modifier d'anciennes. Ces opérations sur la table des descripteurs ont lieu lorsqu'on les rencontre en examinant le texte ("scan"). Ces opérations statiques sont appelées "déclarateurs" pour les distinguer des opérations de génération de code dynamique appelées opérateurs.

Chaque descripteur a un nom symbolique unique et ne doit être référencé qu'après la création d'une entrée (de descripteur) - permettant ainsi une compilation en un passage suivant les besoins. Le nom symbolique du descripteur est utilisé pour toute référence au descripteur à partir de déclarateurs ou d'opérateurs.

Chaque entrée de la table des descripteurs est composée d'un ensemble de champs contenant l'information citée. Ces champs sont "ouverts" c'est-à-dire référençables en BL/1 ou "cachés" lorsqu'ils ne sont pas directement référençables.

Il n'y a pas de raison particulière de référencer les champs cachés. Ils peuvent donc être organisés de manière à convenir au mieux au processeur.

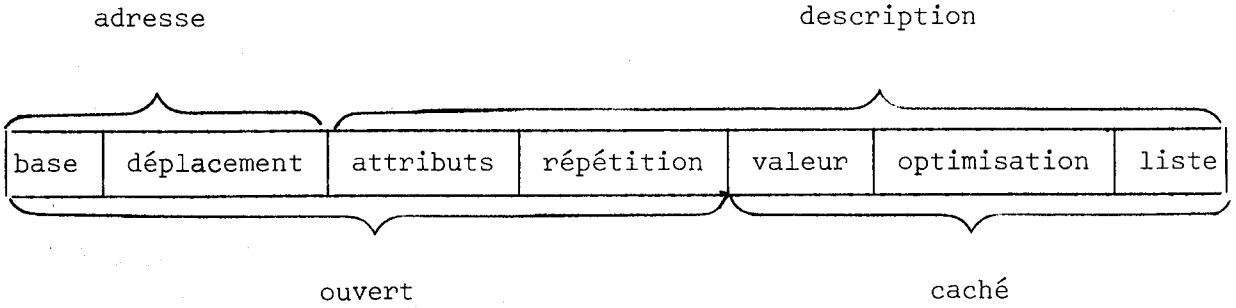
Les champs ouverts, d'autre part, doivent être spécifiés. En particulier un sous-champ d'un bit détermine si le sous-champ suivant est une référence "immédiate" ou "éloignée".



Si "immédiat" est spécifié alors le sous-champ référence contient la valeur littérale de l'objet (si c'est un entier c'est la valeur littérale de cet entier ainsi de suite).

Si "éloigné" est spécifié alors un pointeur de descripteur est à trouver dans le sous-champ référence. Ce pointeur vers une autre entrée de descripteur donne, au moment de la génération, l'information nécessaire pour situer le datum requis par le premier descripteur. Un bit supplémentaire (non figuré ci-après) spécifie si le second descripteur est une valeur ou son adresse. Puisque le second descripteur peut référencer d'autres descripteurs une référence particulière de génération peut utiliser toute une série de descripteurs. Ceci sera plus clair sur des exemples spécifiques.

Considérons maintenant les champs d'une entrée de la table de descripteurs :



2.21 - Champs d'adresse :

D'une manière très semblable à une BASED structure en PL/1 [13] ou à un record en Algol W [32] une adresse BL/1 est considérée comme un couple de composants : une adresse de base et un déplacement entier en fonction de cette base. Ces valeurs (ou références de descripteur pour trouver ces valeurs) sont introduites dans les champs de base et déplacement du descripteur. La composition de la valeur d'une adresse à partir de la paire base-déplacement est faite par une opération "définie pour l'implémentation" qui "ajoute" une adresse à un entier et retourne une adresse. Grâce à cette approche et au fait que la forme interne d'une adresse constante elle-même est laissée indéfinie, il est possible d'adapter BL/1 à une large gamme de schémas d'adressage en machine ou les variations apparaissent seulement au niveau de l'implémentation du processeur et non à celui du langage.

La génération de code pour une instruction de référence à un descripteur devrait être plus claire maintenant. Une fois encore les détails précis dépendront du niveau d'efficacité de l'implémentation et du calculateur utilisé. Le sujet sera discuté plus loin au chapitre 6.

2.22 - Le champ d'attributs -

Une valeur immédiate entrée dans ce champ se compose de différentes parties :

- un indicateur de mode
- un autre d'alignement
- d'entier(s) pour décrire le mode (si nécessaire)
- une valeur ou un pointeur de descripteur donnant la taille de l'unité d'un objet
- un descripteur pointeur de listes pour un groupe de modes.

* L'INDICATEUR DE MODE : est un code pour l'une des dix primitives :

(1)	(2)
adresse	binaire fixe
étiquette	décimal fixe
	binaire flottant
	décimal flottant
espace	caractère
groupe	bit

l'ensemble des codes (2) correspondent aisément à leurs homonymes PL/1. Une "adresse" est une position dans la donnée. Une "étiquette" a pour valeur une position du programme codé. Un "espace" est un objet de donnée qui peut contenir des allocations d'autres objets de données - c'est une généralisation de l'AREA en PL/1. Enfin un "groupe" est une tête de liste d'autres descripteurs ou de listes de descripteurs et par là une forme simplifiée d'une STRUCTURE en PL/1.

* L'INDICATEUR D'ALIGNEMENT spécifie "aligné" ou non "aligné".

* LES ENTIERS QUI DECRIVENT LE MODE s'appliquent seulement à des types numériques. Les modes flottants ont un entier qui définit le nombre maximum de chiffres de la mantisse. Les modes fixes ont un entier supplémentaire indiquant le facteur d'échelle à appliquer à l'objet descripteur.

* LA TAILLE D'UNITE est une valeur déterminant le volume de l'encombrement requis pour un exemplaire de l'objet.

* LE POINTEUR DE DESCRIPTEUR DE LISTE s'applique seulement à un groupe de modes. Le descripteur pointé est la tête d'une liste de descripteurs qui sont subordonnés au descripteur de groupe. Ceci correspond à une définition de structure.

2.23 - Le champ de répétition -

C'est tout simplement le nombre d'exemplaires de l'objet décrit. C'est un sauf pour les tableaux et les chaînes.

2.24 - Le champ valeur -

Si les champs d'adresse du descripteur sont vides, alors ce champ est utilisé pour établir la valeur de l'objet représenté par le descripteur. Dans ce cas le descripteur se comporte comme une constante pour la compilation en cours.

Cependant si les champs d'adresse sont utilisés alors l'entrée est interprétée comme la valeur initiale que l'objet adressé doit avoir au début de l'exécution du programme. Au moment de la génération les champs d'adresse sont utilisés pour situer l'objet.

La valeur d'un descripteur de groupe est définie par un pointeur descripteur placé dans le champ valeur. Ce second descripteur est usuellement la tête de toute une liste de descripteurs dont les champs valeur sont utilisés pour définir les valeurs des descripteurs de la liste du groupe (tous les champs valeurs dans la liste du groupe sont alors ignorés). Les valeurs sont appliquées dans leur ordre d'apparition S'il y a plus de valeurs que de champs, les valeurs supplémentaires sont ignorées. Si par contre, il en manque, les derniers champs ne sont pas initialisés. Cette organisation plutôt singulière est le résultat direct de la manière dont l'attribut INITIAL est défini en PL/1.

Les valeurs des chaînes sont appliquées unité par unité à chaque exemplaire de l'objet d'un descripteur multiple (c'est-à-dire ceux dont le champ de répétition a une valeur supérieure à un). Si la longueur de la chaîne ne correspond pas à la valeur de la répétition alors l'interprétation est celle donnée pour un groupe de valeurs.

D'autre part, pour les valeurs qui ne sont pas des chaînes, tous les exemplaires de l'objet sont initialisés à une même valeur lorsque la répétition est supérieure à un.

2.25 - Le champ d'optimisation -

Il est utilisé par le processeur lorsqu'il range l'information du descripteur particulier ce qui peut être fort utile pour la génération et l'optimisation du code. On pourrait par exemple garder trace des registres principaux où les objets se trouvent couramment ou établir des descripteurs de pointeurs lorsque les valeurs doivent être trouvées indirectement. L'utilisation de ce champ repose complètement sur la nature de l'implémentation et de la machine.

2.26 - Le champ liste -

Il est plutôt utilisé pour joindre le descripteur à une liste de descripteurs. Comme expliqué plus haut cette liste est pointée par un champ d'attribut ou par un autre descripteur et par là devient subordonné à ce descripteur. Une imbrication adéquate des groupes produit n'importe quel arbre de descripteurs.

2.3 - Les instructions en BL/1 -

Toute instruction a la forme d'un mot clef suivi par un nombre fixe de paramètres. Certains paramètres peuvent être préfixés par un modificateur mais c'est la limite de leur complexité syntaxique. Cette simplicité est le résultat conscient que tout le lourd travail doit être fait autant que possible par la traduction en macro.

Il y a deux formes fondamentales de paramètres :

- représentations littérales (constantes)
- références à d'autres descripteurs.

La forme exacte d'un littéral importe peu. Il est suffisant de savoir qu'une forme plausible existe pour les types primitifs : caractère, bit, valeurs numériques et valeurs de champs d'attributs. A la différence de langages type ALGOL 68 et PL/1 le mode d'une constante n'est pas défini par sa forme mais par le contexte dans lequel il se trouve. Si une constante apparaît par exemple dans le champ déplacement d'une instruction BL/1 ce doit être un entier c'est-à-dire binaire fixe de précision par défaut.

Si le paramètre est un descripteur référencé, sa forme exacte variera légèrement en fonction de l'endroit où il apparaît dans une instruction statique (un déclarateur) ou dynamique (un opérateur). Par conséquent, ces formes vont être discutées cas par cas dans les sous-sections qui suivent.

2.31 - Déclarateurs -

Un déclarateur est une instruction qui crée un nouveau descripteur ou qui en modifie un. Un paramètre de déclarateur peut référencer un autre descripteur de deux manières :

- soit en sélectant un champ à copier à partir de ce descripteur,
- soit en créant un pointeur vers un autre descripteur.

Les sélections sont spécifiées par un modificateur suivi par un nom de descripteur. Il y a quatre modificateurs possibles, un par champ ouvert :

base - champ de base

off - champ de déplacement

attr - champ d'attribut

rep - champ de répétition

Si un pointeur vers un autre champ est à créer, le pointeur est destiné à indiquer soit la valeur de l'objet décrit soit son adresse. Dans le premier cas seul le nom de descripteur est donné comme paramètre et dans l'autre le nom précédé par le modificateur adr est utilisé.

Une désignation de paramètre de déclareur destiné à remplir un certain champ de descripteur doit être conforme au mode exigé par ce champ. Aussi les champs ci-dessous doivent-ils être spécifiés par la construction syntaxique donnée :

champ base - <entrée-adresse>

champ déplacement - <entrée-entière>

champ attribut - <entrée-attribut>

champ répétition - <entrée-entière>

où

entrée-adresse → adresse-descripteur

| 'base' nom-descripteur | 'adr' nom-descripteur

entrée-attribut → spec-attribut

| 'attr' nom-descripteur

entrée-entière → entier-descripteur | décimal-littéral

| 'off' nom-descripteur | 'rep' nom-descripteur

Une <adresse-descripteur> est un nom de descripteur qui réfère une entrée descripteur ayant l'attribut de mode d'une adresse.

Un <entier-descripteur> est un nom de descripteur qui réfère une entrée de descripteur ayant un attribut de mode binaire-fixe et un facteur d'échelle zéro.

2.311 - Déclarations des formes des données -

Nous allons passer en revue quelques uns des déclarateurs :

2.3111 - describe nom @ base, déplacement : attributs, répétition.

Un nouveau descripteur est créé avec un "nom" symbolique et des champs remplis par les valeurs spécifiques des entrées correspondantes. Le champ valeur est nul et la valeur taille est calculée suivant les attributs spécifiques.

Exemple :

```
describe Ai @ A, t : attr A, 1
multiply I * size Ai → t
pass x → Ai
```

est la compilation de l'affectation PL/1 A(I) = X. (Un descripteur "Ai" est défini comme la tème adresse de l'unité d'un tableau A. Pour trouver ce $i^{ème}$ élément, on multiplie la taille de l'unité par i et on place le résultat dans un entier t temporaire. La valeur x peut alors être placée en A(I)).

2.3112 - sketch nom : attribut, répétition.

C'est la même fonctions que describe mais les champs base et déplacement ne sont pas spécifiés. Si sketch est hors d'une déclaration layout (voir ci-dessous) alors le champ déplacement reçoit la valeur zéro et l'adresse de base est nulle. Sinon le layout contrôle la valeur à leur donner.

Exemple : sketch A : axb 31 + 0, 100

est DECLARE A(100) FIXED BINARY (31, 0) de PL/1. (Produit un descripteur pour un tableau A de 100 nombres binaires fixes, alignés de précision (31, 0)).

2.3113 - layout nom : répétition

frame

Ces deux déclarateurs renferment une structure de bloc de la même manière que struct (...) d'Algol 68. Un descripteur est créé, on donne une valeur au champ de répétition comme il est indiqué et l'attribut est celui d'un groupe dont un pointeur vers la liste de descripteurs définis dans le bloc. Base et déplacement sont ceux de sketch. A la création de chaque nouveau descripteur le champ liste du descripteur précédent devient un pointeur vers ce dernier ; lorsque le frame correspondant est trouvé tous les descripteurs ont reçu des adresses : les bases se réfèrent à l'objet adresse du groupe de descripteur et les déplacements constitués suivant la position relative et les déplacements constitués suivant la position relative des membres en prenant en compte leur alignement, mode et multiplicité respectifs. Pour cela, l'effet est de lier entre eux les membres du groupe.

Exemple :

```
layout A : 100
  sketch S : c, 20
  layout T : 1
    sketch U : a, 1
    sketch V : a, 1
  frame
frame
```

C'est DECLARE 1 A (100),

2 S CHAR (20),

2 T

3 U PTR, 3 V PTR ;

Une note d'optimisation : on peut discuter de l'établissement des déplacements des descripteurs de membre à établir lorsqu'on les rencontre plutôt que d'attendre le frame qui termine. Ceci est vrai si on peut accepter une allocation moins optimale sur des machines pour lesquelles l'alignement mémoire est important. Sur de telles machines, il est possible d'aller à l'autre extrême et de réordonner les membres descripteurs pour atteindre l'efficacité maximum possible. Pour diverses raisons ceci est sujet à quelques contraintes globales sur le programme et ne peut être effectué dans une implémentation en un passage. En tout cas, cette option est ouverte et laissée au choix de l'implémenteur - en particulier l'algorithme de l'alignement.

2.3114 - link descripteur to adresse

Ceci insère dans le champ base du premier descripteur (qui doit être déjà défini) un pointeur vers le second descripteur. Ainsi le groupe ou le membre simple représenté par le premier descripteur est toujours trouvé relativement à l'adresse représentant le second. C'est essentiellement la même fonction donnée par le qualificateur d'adresse (—>) en PL/1.

Exemple : link A to P.

(Ceci aurait été produit si la précédente structure avait eu l'attribut BASED (P)).

2.3115 - value descripteur = constante.

Le déclarateur value complète le champ valeur d'un descripteur déjà déclaré.

Dans le cas d'un groupe descripteur un pointeur de référence est fabriqué pour le groupe désigné. Comme expliqué plus haut aucune valeur de champ d'entrée n'est créée. C'est plutôt une correspondance logique qui est établie.

Exemple : value S = "ALORS ?"

(Ce serait produit si la chaîne de caractère précédente était INITIAL ("ALORS ?") de PL/1).

2.3116 - constant nom = valeur : attributs, répétition.

Tandis que le déclarateur value place des valeurs dans le champ valeur du descripteur, constant créé tout un descripteur ayant valeur, attributs et répétition comme il est spécifié. Les deux champs base/déplacement ne sont jamais créés par le langage, puisque lorsque le descripteur est référencé, les contenus du champ valeur sont toujours retournés sans référence à leurs champs d'adresse. (A contrario de value). Que la constante occupe ou non de l'espace dépend de la situation et de l'implémentation le compilateur n'ayant rien de plus à y faire. (Dans l'implémentation plusieurs noms peuvent pointer vers la même valeur).

La manipulation des champs valeur est très semblable à celle de value sauf pour l'indicateur nul (o) utilisé pour spécifier des constantes étiquettes. Cela laisse le champ valeur vidé. On s'attend alors à le garnir plus tard par label.

2.3117 - label descripteur

Il prend le descripteur prédéfini (usuellement par sketch ou constant) et insère comme valeur l'adresse courante du code en cours de génération. Cette valeur est nécessairement une constante et pourrait servir de valeur initialisante (comme une variable étiquette PL/1).

Exemple : constant L = o : ℓ, 1

```
          ...
label L
          ...
goto L
```

Le descripteur d'étiquette L est créé par constant ; une adresse courante comme valeur est donné à label et le flot de contrôle est transféré à cette adresse du goto. A noter que constant doit apparaître en premier lieu).

Note d'optimisation : En insistant sur la désignation de l'étiquette avant son usage, on permet de circonvenir au problème classique des références "en avant" dans un processus à un passage. Evidemment cela ne produira pas un code optimal mais l'occasion est donnée de produire une implémentation "rapide-et-sale" si on le désire. Il n'est pas exclu d'utiliser une technique en deux passages pour un code meilleur - il va de soi.

2.312 - Allocateurs statiques -

Jusqu'ici les seuls déclarateurs considérés établissent forme et structure des éléments de données sans s'occuper où et quand trouver ces éléments en mémoire. La séparation de ces deux concepts est délibérée : l'un des avantages des descripteurs est de permettre une décomposition des déclarations de données en ces notions primitives et séparées.

En BL/1 les allocations sont soit "statiques" c'est-à-dire fixées à la compilation, soit "dynamiques" c'est-à-dire déterminées pendant l'exécution. Par définition un allocateur BL/1 est soit un déclareur soit un opérateur.

Deux allocateurs statiques seront décrits ici : share et fasten. Ils correspondent à peu près aux attributs relatifs à la mémoire en PL/1 : STATIC EXTERNAL et STATIC INTERNAL.

2.3121 - fasten descripteur

Il prend le descripteur (qui peut être simple ou descripteur de groupe auquel sont subordonnés d'autres descripteurs) pour déterminer le montant adéquat d'espace à allouer au programme en cours de génération. L'adresse de cet espace est entré dans le champ de ce descripteur en valeur immédiate (constante). Tous les champs valeurs correspondants sont examinés (comme décrit dans le paragraphe sur les entrées des champs valeur) et les entrées constantes sont préparées pour le code généré. A noter que tous les paramètres affectant le montant d'espace alloué (par exemple les champs répétitions) doivent être fixés quand fasten est trouvé. A noter de plus qu'il n'y a pas de moyen direct d'accéder à ces adresses sauf à partir du segment programme courant (équivalent à EXTERNAL PROCEDURE de PL/1).

Exemple : fasten A
(pour une structure dont les attributs sont STATIC INTERNAL).

2.3122 - share descripteur as nom-externe.

De la même manière que le fait fasten, share dispose l'espace à allouer pour l'objet du descripteur. La distinction en est que l'espace est aussi disponible à d'autres segments de programme. Il est identifié au moyen du nom-externe. Les autres segments de programme utilisant la même donnée doivent avoir par hypothèse des allocateurs share similaires (la sous-structure peut être différente mais rappeler les mêmes réservations).

Les champs valeurs sont rassemblés comme pour fasten et sont préévalués au chargement. Les champs de base/déplacement du descripteur sont établis pour trouver l'adresse de la donnée lors de l'exécution.

Exemple : share A as "A"
(pour une structure STATIC EXTERNAL)

note d'optimisation : La question d'identification des allocations à partir de différents segments de programme et comment le système est supposé les traiter, est laissée complètement ouverte. Une solution simple (pour le chargeur) est de référencer toutes les adresses d'allocations partagées via une table des adresses de base (par les adresses cachées dans les descripteurs), ensuite le chargeur n'a besoin que de positionner les entrées qui correspondent à des noms externes identiques. Ceci produit un code plus compliqué que nécessaire. Une façon de produire des programmes "liés" plus efficaces serait de marquer les références machine de données lorsqu'elles interviennent et de faire insérer au chargeur les adresses adéquates dans le code.

2.32 - Opérateurs -

Rappelons qu'un opérateur BL/1 est une instruction pour laquelle du code machine est effectivement généré. Les paramètres de ces opérateurs sont : ou bien des pointeurs vers des descripteurs, ou bien des sélections de champs copiés à partir d'autres descripteurs. Les sélections autorisées sont :

base — champ de base

off — champ déplacement

rep — champ répétition

(une sélection d'attribut n'aurait pas de sens car un paramètre d'opérateur ne pourrait jamais avoir ce mode).

Il y a trois sortes de pointeurs de descripteur au lieu de deux :

valeur (nom de descripteur seul)
adresse (adr suivi par un nom de descripteur)
taille (size suivi par un nom de descripteur)

Les pointeurs de valeur et d'adresse sont définis comme pour un déclarateur. Le pointeur de taille référence un descripteur qui, à la génération, devra extraire du champ attribut de ce descripteur un entier qui est la taille calculée d'un seul exemplaire de l'objet de donnée. Le pointeur de taille est utile en établissant le "multiplicateur" d'un indice de tableau. (On peut raisonnablement se demander pourquoi le pointeur de taille n'est pas permis également comme paramètre de déclarateur. De fait, la seule raison est que l'anomalie n'était pas évidente dans la formulation primitive de la définition. En tout état de cause je n'en ai jamais trouvé un usage comme paramètre de déclarateur).

Dans certains cas l'usage d'autre chose qu'un pointeur de valeur comme paramètre compliquerait inutilement la définition de l'opérateur BL/1. Par exemple, il ne serait pas utile d'avoir un pointeur taille comme but d'une affectation (la taille d'un objet est établie seulement par son déclareur).

2.321 - Manipulations courantes -

Le coeur de tout langage de programmation est son aptitude à manipuler des objets (des données) et le programme lui-même. En BL/1 ceci revêt l'aspect d'opérations binaires ou unaires standardisées dans la grande majorité des cas. Dans ce qui suit nous les décrivons "comme elles viennent" en commentant leurs fonctions particulières - quoique ces fonctions soient "transparentes". Il y a des "idiotismes" bien sûr comme c'est toujours le cas pour certaines fonctions d'un langage et un nouveau déclarateur, shape, qui se rapporte à la manière curieuse de définir des objets temporaires comme l'exige les règles de l'arithmétique PL/1.

2.3211 - goto étiquette
goif chaînedebits to étiquette

Ces deux formes transfèrent le flot de contrôle à la position indiquée par le descripteur étiquette de l'objet. Le goif transfère ce contrôle seulement si l'un au moins des bits du descripteur est positionné.

2.3212 - pass descripteur$_1$ to descripteur$_2$.

Cet opérateur prend l'objet du descripteur$_1$ et la place dans le descripteur$_2$. Les champs répétition et attribut des deux descripteurs doivent être identiques et constants, les répétitions peuvent être variables mais égales à l'exécution.

2.3213 - change descripteur$_1$ to descripteur$_2$.

Cet opérateur n'est défini que pour des descripteurs de type chaîne ou de type numérique. Son propos est de fournir une méthode de conversion entre types de base qui soit conforme aux règles de conversion de PL/1. Ces règles sont hautement dépendantes de l'implémentation et pour cette raison reléguées au niveau de l'implémentation de BL/1. Le traducteur n'est pas concerné par les détails de conversion s'il utilise cet opérateur et le déclarateur shape.

Les descripteurs numériques doivent avoir un facteur de répétition de un tandis que les descripteurs de chaîne peuvent contenir en répétition un pointeur vers un descripteur ou une constante.

Si le descripteur destination est une chaîne alors son champ taille reçoit la longueur appropriée si variable était indiqué ou bien la chaîne elle-même est cadrée à droite si le champ taille indique une constante. Le complément ("padding") est fait de zéros ou de blancs suivant le cas. De même, la longueur d'une chaîne pour le descripteur source se trouve dans son champ taille.

Exemple : Considérer le texte PL/1 et le code BL/1 équivalent :

DECLARE C CHAR(10) VARYING, X FIXED DECIMAL (4, -3) INITIAL (1234.0E5) C = X ;	sketch C_ℓ : attr protoxb, 1 sketch C : c, C_ℓ sketch X : xd 4-3,1 value X : 1234E+5 change X to C

(Après exécution C a la valeur " 1234F+5" et C_ℓ la valeur 8).

2.3214 - opérateur descripteur$_1$, descripteur$_2$ → descripteur$_R$

opérateur descripteur → descripteur$_R$

Ce sont les opérations "standardisées" et courantes mentionnées au début de ce paragraphe. En général les objets désignés par descripteur$_1$ et descripteur$_2$ sont combinés en fonction du mot-clef de l'opérateur et le résultat est placé dans l'objet désigné par le descripteur destination R. A noter que les affectations sont toujours effectuées de gauche à droite.

L'exécution des opérations en "deux temps trois mouvements" :

- pré-conversion de l'un (ou des deux) opérandes en unités aux attributs compatibles (en fonction des règles PL/1 pour cet opérateur)
- exécution de l'opération elle-même
- post-conversion (éventuelle) pour se conformer aux attributs du descripteur destination.

Les premier et dernier pas peuvent être également effectués par change si on peut déterminer les attributs. Dans les cas courants il est inutile de se préoccuper de ce détail.

Les descripteurs numériques doivent toujours avoir un facteur de répétition de 1 et les chaînes sont traitées comme pour change.

Voici une liste d'opérations individuelles (leur signification en français est évidente) :

ARITHMETIQUES :

add substract multiply divide modulus ceiling floor minus	Elles travaillent uniquement pour des descripteurs numériques en exécutant les opérations mathématiques qu'elles indiquent. Modulus est le reste d'une division, ceiling et floor sont celles définies par Iverson [ ]. Minus est le préfixe "-" d'une expression.

DE COMPARAISON :

equals larger smaller	Pour comparer deux opérandes et produire un bit unique. Larger et smaller sont définies pour des nombres et des chaînes, equals également ainsi que des étiquettes et des pointeurs.

BOOLEENNES :

and or not	Définies uniquement pour des descripteurs bit. Le résultat en est les chaînes de bits obtenues par application des règles classiques que leur nom représente en mathématiques.

D'ARRONDIS :

X-round F-round	X-round s'applique aux descripteurs à point fixe et F-... à flottant. Le second opérande de X-round est un entier donnant le nombre de troncature, F-round arrondit toujours le dernier chiffre de la mantisse.

note d'optimisation : Si les précédentes instructions sont exécutables, il

est évident que lorsque les deux opérandes sont constants le résultat le sera aussi. Le processeur BL/1 peut interpréter ces résultats et améliorer l'exécution du programme généré.

2.3215 - shape nom to opérateur descripteur$_1$ & descripteur$_2$.

La décomposition de l'expression PL/1 par le traducteur donne une série d'opérateurs unaires ou binaires en BL/1. Ceci implique la création de résultats temporaires (ou intermédiaires) et puisque BL/1 insiste sur la description de tous les objets, le problème se pose de choisir les attributs de ces objets intermédiaires. Plutôt que de surcharger le traducteur par un ensemble complexe de règles de conversion PL/1 (qui doit être dupliqué en tous cas en BL/1) ce déclarateur est spécialement défini pour créer un descripteur dont le champ attribut est propre à recevoir les résultats de l'opérateur indiqué. Ce dernier opère sur les deux descripteurs comme opérandes.

Exemple : l'instruction PL/1 :

```
A = B / (C - D)
```

est traduite en :

```
shape t to substract C & D
substract C, D → t
divide B, t → A
```

(S. SCHUMAN a remarqué qu'en fait shape pouvait être éliminé si les résultats intermédiaires peuvent être déclarés implicitement. Dans l'exemple ci-dessus l'instruction substract devrait avoir la même fonction déclarative que celle indiquée explicitement par l'instruction shape).

2.322 - Allocations dynamiques -

Il y a trois types d'allocateurs dynamiques qui effectuent tous la même fonction de base pour obtenir de l'espace (mémoire) pendant l'exécution d'un programme. Le processus général en sera discuté tout d'abord puis les descriptions spécifiques de chaque type seront passées en revue.

Strictement dit, il y a des allocateurs (stack, heap et/ou clump) et des dé-allocateurs (unstack, unheap et/ou unclump). Les allocations sont faites à l'intérieur d'espaces (décrits par les descripteurs d'espace). Puisque tout espace peut contenir des données de type espace, des sous-allocations et d'autres imbrications sont possibles. Le programme commence avec un grand espace défini par le prélude (appelé "sysarea") qui représente toutes les zones de données inutilisées. Les allocations sont effectuées jusqu'à l'épuisement de l'espace total où la condition error est signalée pour la tâche courante. On suppose que tous les espaces ont une sous-structure cachée (comme défini plus haut) suffisante pour localiser l'allocation courante ou pour déterminer qu'elle est vide. L'espace constant défini par le prélude de nom "espace nul" n'est pas allouable dedans. Si cette constante est affectée à un espace quelconque alors cet espace est non réservable.

Le descripteur d'un objet à allouer doit avoir un pointeur d'adresse dans son champ base. L'objet du descripteur de base (qui doit être une seule adresse) s'attache l'allocation de l'objet original. L'inverse est vrai pour la dé-allocation : l'adresse de base est utilisée pour localiser l'objet à libérer et le volume d'espace est rendu en fonction de la taille (champ taille) du descripteur original.

Les trois types d'allocations ne peuvent être entre mélés dans le même espace. Si on commence l'allocation d'un espace par stack on ne peut utiliser un allocateur différent dans le même espace. En voici sa description :

2.3221 - stack descripteur in espace
unstack descripteur from space

L'opération d'empilement (stacking) impose au programme l'obligation d'exécuter stack et unstack dans l'ordre DEPS (Dernier Entré, Premier Sorti - c'est le classique LIFO : Last In, First Out). Le système doit, en retour, sauver la dernière valeur de l'adresse de base à l'empilement et la restaurer en désempilant. Si le volume d'espace disponible dans l'objet désigné par le descripteur d'espace n'est pas adéquat, alors l'espace est automatiquement étendu (au moyen de l'allocation cachée en

en "sysarea"). Par conséquent, l'extension est récupérée lorsqu'on n'en a plus besoin.

Exemple : stack A in progarea
(si A est un tableau AUTOMATIC ceci allouera l'espace nécessaire)

2.3222 - heap descripteur in espace
unheap descripteur from espace

C'est la même opération que stack/unstack sauf pour les adresses de bases qui ne sont ni sauvegardées ni restaurées et corollairement pour lesquelles l'ordre importe peu. Pratiquement ceci nécessite une routine d'utilisation de mémoire libre que l'implémenteur définira. Extension/Retraction automatiques d'espace fonctionnent comme pour stack.

Exemple : heap A in sysarea
(C'est la traduction de l'instruction PL/1 :
ALLOCATE A
où A a l'attribut mémoire BASED)

2.3223 - clump descripteur in espace
unclump descripteur from espace

Ces opérations fonctionnement comme stack/unstack à la seule exception que lorsque l'espace disponible est épuisé, une condition de zone ("AREA condition") est signalée au lieu de l'extension automatique d'espace.

Exemple : clump A in PA
(Supposant A défini comme au § 2.3222 et PA par DECLARE PA AREA
alors cette opération est la traduction de
ALLOCATE A IN (PA)).

2.4 - L'usage des descripteurs -

De l'aveu de certains, il est plutôt difficile de saisir la signification des descripteurs BL/1 sans avoir vu sur un exemple le rôle qu'ils jouent dans la génération de code. La discussion suivante est destinée à éclairer le pourquoi et le comment de ces descripteurs pour différents actes de traduction.

Pour commencer, supposons l'existence des deux déclarations PL/1 :

```
DECLARE (P, Q) POINTER STATIC INTERNAL ;
```

Dans le code BL/1 correspondant les descripteurs de P et Q seront créés par les instructions :

```
sketch P : a, 1 ;
sketch Q : a, 1 ;
```

Les descripteurs auront la forme :

	base	déplacement	attribut	répétition	liste
P :	_	0	a	1	_
Q :	_	0	a	1	_

Cinq des sept champs sont représentés. Ce seront les seuls qui nous intéresseront dans les exemples suivants. A noter que le champ base n'est pas encore établi.

Puisque les deux pointeurs sont STATIC INTERNAL ils doivent recevoir des positions fixes dans l'espace des données du programme généré. On l'indique par :

```
fasten P ;
fasten Q ;
```

ce qui impose au compilateur BL/1 l'établissement des adresses P et Q et le placement de celles-ci dans les descripteurs correspondants :

P :

αP	0	a	1	-

Q :

αQ	0	a	1	-

αP, αQ sont des valeurs représentant les adresses de P, Q.

Un cas plus compliqué est la déclaration d'une structure :

```
DECLARE 1 S BASED(P)
          2 A CHAR(8),
          2 B POINTER ;
```

Les descripteurs correspondant à S, A et B sont créés par les instructions :

```
layout S : 1 ;
   sketch A : c, 8 ;
   sketch B : a, 1 ;
frame ;
```

L'instruction layout pour S crée le descripteur S :

S :

—	0	g ( )	1	—

Le premier sketch construit le descripteur A et lui ajoute un pointeur à partir du champ attribut de S :

A :

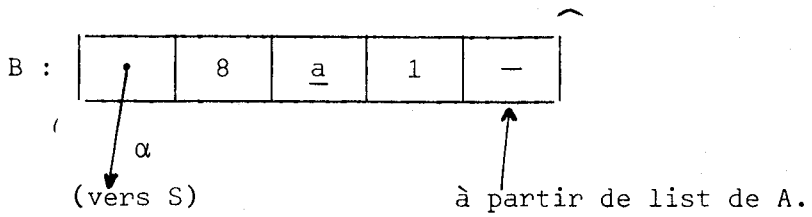
•	0	c	8	—

α (vers S)

à partir d'attr de S

A noter que le pointeur "à partir" du champ base de A (marqué α) est un pointeur d'adresse (A a la même adresse que S plus le déplacement qui est zéro dans ce cas).

Le second sketch construit le descripteur B et remplit le champ liste de A par un pointeur vers B :



Un compte courant de la longueur de S est gardé pour garnir correctement le champ déplacement. Le déplacement de B est huit dans ce cas puisque ceci prend en compte l'espace occupé par A.

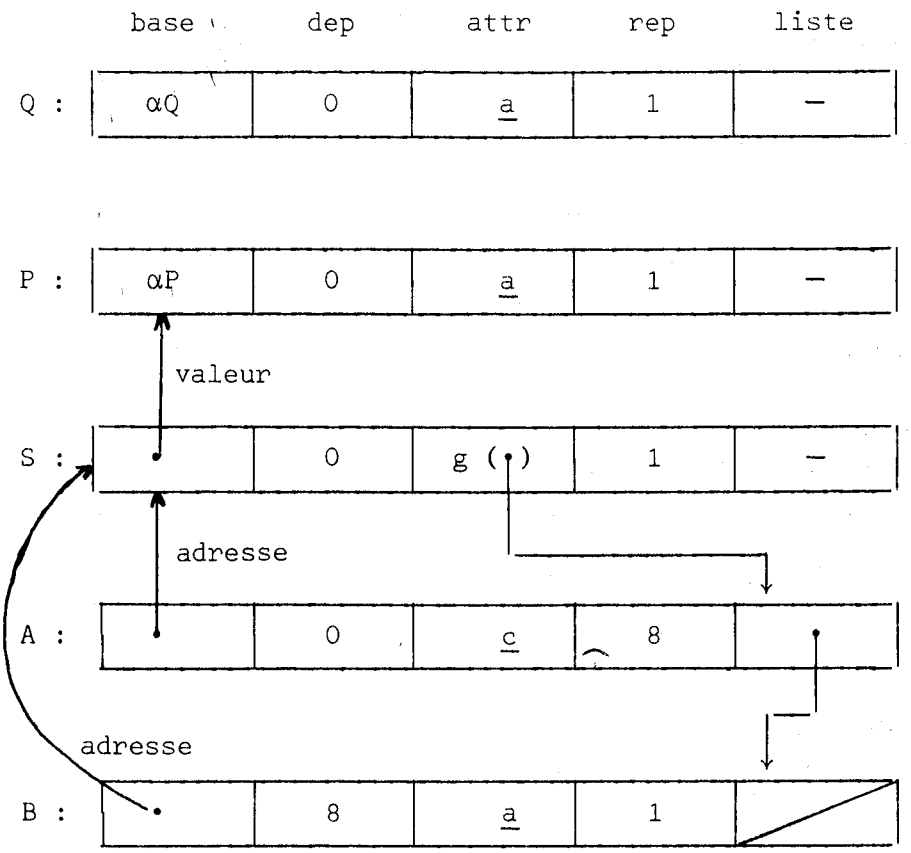
Finalement frame garnit le champ liste du dernier descripteur trouvé (B dans le cas présent) par un marqueur de fin de liste. Ceci ferme la structure.

S a l'attribut mémoire BASED (P). Ceci est indiqué par l'instruction BL/1 :

```
link S to P ;
```

qui garnit le champ base du descripteur S par un pointeur-valeur vers P (S est localisé par l'adresse contenue en P et non par l'adresse de P)

Après le traitement de tous ces déclarateurs, l'état final de la table des descripteurs est :



Supposons l'instruction PL/1 à traduire :

Q = P → S • B ;

L'équivalent BL/1 est tout simplement :

pass B to Q ;

Afin de compiler cette instruction la valeur de B et l'adresse de Q doivent être déterminés par le compilateur BL/1. La fonction d'accès pour ces deux variables est décrite complètement par le contenu de la table des descripteurs.

En général ceci implique de suivre un arbre de pointeurs et de générer des instructions en revenant. Dans notre cas particulier voilà ce qui se passe :

. l'adresse de B est l'adresse de S plus huit
. l'adresse de S est la valeur de P plus zéro
. l'adresse de P est la constante αP
. la valeur de P est placée dans le registre 1 en générant

L 1, αP

. ceci établit l'adresse de S dans ce registre 1
. l'adresse de B est 8 plus le contenu de ce registre 1
. la valeur de B est placé dans le registre 2 en générant

L 2, 8(1)

. l'adresse de Q est la constante αQ
. l'affectation est terminée en générant

ST 2, αQ

Remarque :

D. SERAIN [21] a montré comment toute fonction d'accès pouvait être générée en utilisant plusieurs sous-programmes systématiques qui manipulent la table des descripteurs.

2.5 - Un court exemple de programme -

Le problème réel de la traduction de PL/1 en BL/1 sera abordé plus tard (ch. 3). Il est cependant utile d'examiner ici un exemple de programme PL/1 en regard de l'équivalent BL/1 afin de donner une idée de ce que l'on entend par le propos poursuivi, à savoir que "tout programme PL/1 a son équivalent sémantique en BL/1".

Toutefois puisque BL/1 est particulièrement conçu pour manipuler aisément les fonctions d'accès aux données, l'exemple qu'on présentera fera largement mention de cette propriété et donnera également la façon de traiter les paramètres PL/1 (c'est-à-dire l'appel par adresse).

Supposons la fonction PL/1 écrite pour calculer les coefficients du binôme en utilisant la formule (très inefficace) :

$$\binom{n}{m} = \binom{n-1}{m-1} \frac{n}{m} \quad \text{(pour } m > 0\text{)}$$

avec $\binom{n}{0} = 1$

Le programme requis est :

```
BC : PROCEDURE (N, M) RECURSIVE RETURNS (FIXED BINARY (15, 0)) ;
     DECLARE (N, M) FIXED BINARY (15, 0) ;
     IF M > 0
     THEN RETURN  (BC (N-1, M-1) * N/M) ;
     ELSE RETURN (1) ;
     END ;
```

Nous allons considérer maintenant quel est le programme équivalent écrit en BL/1. Le premier problème est de décrire correctement les paramètres formels qu'utilise la fonction. A partir des déclarations de N et M nous avons directement

```
sketch N : xb 15 + 0 ;
sketch M : xb 15 + 0 ;
```

Ceci spécifie plutôt la création de deux descripteurs N et M, qui auront la même forme d'attributs équivalents à FIXED BINARY (15, 0). Les adresses des valeurs doivent cependant être spécifiées. En PL/1 l'adresse d'un paramètre est passée et non pas la valeur ou le nom. Ceci signifie qu'une structure doit être définie qui prend en compte les adresses des paramètres dans un ordre prédéterminé (pour simplifier les choses plus tard, la valeur du résultat de la fonction sera aussi incluse dans ce bloc) :

```
layout paramètres ;
   sketch valfonc : xb 15 + 0 ;
   sketch Nadr : a ;
   sketch Madr : a ;
frame ;
```

Ceci crée les nouveaux descripteurs : "paramètres", "valfonc", "Nadr", "Madr", tous sont liés dans un groupe BL/1. Il est maintenant nécessaire de spécifier que la valeur de N se trouve par l'adresse décrite par Nadr (de même Madr pour M).

```
link N to Nadr ;
link M to Madr ;
```

Ces deux instructions détermineront les champs base et déplacement des descripteurs M et N.

Les champs d'adresse du descripteur de paramètres ne sont pas encore résolus. Puisque la procédure est récursive cette adresse doit être sauvegardée dans le bloc mémoire local de l'appel courant. Le bloc est défini par :

```
layout bloc ;
   sketch adrparam : a ;
   sketch adrretour : a ;
   sketch temp1 : xb 15 + 0 ;
   sketch temp2 : xb 15 + 0 ;
frame ;
```

où "adrparam" est l'adresse où trouver le bloc paramètre, "adrretour" range la position de retour dans la procédure appelante, et les temporaires "temp1", "temp2" sont des zones de travail pour l'appel courant. Les champs adresse du descripteur "paramètres" sont créés par l'instruction :

```
link paramètres to adrparam ;
```

Maintenant où trouver le bloc ? Il est repéré à partir du niveau n° 1 du "display" défini par :

```
layout display ;
   sketch niveau1 : a ;
frame ;
link bloc to level1 ;
```

Cela peut sembler ne jamais en finir puisque l'adresse de display doit être maintenant spécifiée. Afin de s'arrêter quelque part, un descripteur "basedisplay" est supposé défini dans le prélude standard de l'implémentation. L'objet du descripteur "display" est trouvé relativement à cette base :

```
link display to basedisplay ;
```

La nature du nom réservé "basedisplay" n'est spécifiée nulle part dans un programme particulier. On laissera à une implémentation le soin de décider s'il faut placer cette valeur en registre central ou bien dans une cellule de mémoire. Les aspects spéciaux de "basedisplay" sont notés quelque part dans le mécanisme interne, c'est-à-dire dans le champ optimisation du descripteur. La faculté de décider pour une implémentation où placer les valeurs de descripteurs, en mémoire centrale ou dans des registres n'est pas limitée aux descripteurs définis au prélude mais c'est particulièrement utile dans ce cas.

Une fois les données du programme décrites, il est maintenant possible de considérer la traduction BL/1 du corps de la procédure. Pour commencer, le point d'entrée BC doit être défini par l'instruction :

label BC ;

(pour suivre les paragraphes qui précédent, on devrait nécessairement définir BC comme le descripteur d'une constante étiquette en utilisant l'instruction constant. Ceci est généralement redondant d'où cette omission pour simplifier - on combinera les aspects déclaratifs et impératifs en une seule instruction).

Nous supposerons que la procédure est activée en transférant le contrôle à la position indiquée par la valeur de BC. A ce point, deux informations doivent être disponible pour l'exécution : l'adresse du bloc de paramètres et la position de l'instruction où le contrôle revient après l'exécution de la procédure. La stratégie utilisant des descripteurs prédéfinis par un prélude peut être exploitée encore ici. On trouve les valeurs aux positions décrites par "regparam" et "regretour" respectivement.

La première action de la procédure consiste à allouer de l'espace pour le bloc local de mémoire et sauvegarder ces deux valeurs :

```
stack bloc in pile ;
pass regparam → adrparam ;
pass regretour → adrretour ;
```

L'instruction stack donne un espace conformément à la zone décrite par le descripteur d'espace "pile". Un effet de bord utile permettra de placer l'adresse de base des zones allouées dans le champ valeur de "niveau1", descripteur pointé par le champ base du descripteur "bloc". Les deux affectations qui sauvegardent les adresses des paramètres et de retour peuvent ensuite être sauvegardées correctement.

L'instruction IF peut maintenant être générée. Le test

```
IF  M > 0
```

devient

```
constant zéro = 0 : xd 1 + 0 ;
larger zéro, M → pasplusgrand
go-if pasplusgrand to étiquette2 ;
```

A noter qu'un descripteur pour la constante zéro a du être créé (avec les attributs FIXED DECIMAL (1, 0)) puisque les arguments de l'instruction larger doivent être des descripteurs. A l'implémentation le choix est libre : soit un emplacement mémoire pour cette valeur soit aucune réservation si on peut l'éviter sur une machine particulière. Le descripteur de bit "pas plus grand" devrait être défini avant usage mais sa nature est suffisamment claire dans ce contexte. Une définition par défaut de ce type devrait être probablement une addition utile au langage puisque la valeur serait située dans le bloc mémoire local même si ce n'est pas justifié pour la machine particulière utilisée.

La clause

```
THEN RETURN (BC (N-1, M-1) * N/M) ;
```

est plus compliquée à cause de l'expression entre parenthèses évaluée d'abord puis placée en "valfonc" pour cet appel. Afin d'effectuer ceci, la fonction BC doit être appelée récursivement avec de nouveaux arguments :

```
constant un = 1 : xd 1 + 0 ;
substract N, un → temp1 ;
substract M, un → temp2 ;
```

(A noter en passant que les références à N et M exigent beaucoup de traitements, puisque cela implique de suivre l'adresse de chaînage par Nadr, Madr, paramètres, adrparam, bloc, niveau1, niveau2, display et basedisplay).

Un bloc contenant les adresses réelles des paramètres et un espace pour la valeur de la fonction doit être préparé maintenant pour la fonction appelée :

```
layout paramreel
   sketch valretournée : xb 15 + 0 ;
   sketch adrarg1 : a ;
   sketch adrag2 : a ;
frame ;
link paramreel to temp3 ;
```

Les descriptions des paramètres réels doivent suivre bien sûr celles des paramètres formels. Le bloc peut être alloué maintenant et rempli par :

```
stack paramreel  in pile ;
pass adr temp1 → adrarg1 ;
pass adr temp2 → adrarg2 ;
```

Une fois l'adresse du bloc paramètre et les registres d'adresse de retour chargés,la procédure peut être appelée :

```
pass temp3 → regparam ;
pass etiquette1 → regretour ;
goto BC ;
label etiquette1 ;
```

La valeur de "temp3" pointe le bloc de paramètre réel après exécution de l'instruction stack. Le descripteur "label1" fournit le point de retour après la fin de la fonction BC au plus bas niveau.

Après l'appel "valretournée" contiendra le résultat de la fonction. On peut utiliser cela pour calculer le reste de l'expression :

```
multiply valretournée, N → BCfois N ;
divide BCfoisN, M → valfonc ;
```

Puisqu'on n'a plus besoin des paramètres réels de bloc on peut les abandonner ici :

```
unstack paramreels from pile ;
```

Pour sauvegarder un peu l'espace, on effectue un transfert vers un point de retour commun :

```
goto pointretour ;
```

La clause ELSE est bien plus simple :

```
label etiquette2 ;
change un to valretournée ;
```

L'instruction change est utilisée pour forcer la conversion de la valeur FIXED DECIMAL (1, 0) en valeur FIXED BINARY (15, 0). Pour une implémentation optimisée aucune génération ne devrait être effectuée. (Remarquer comment cet ennui au niveau PL/1 a filtré au niveau du langage de base).

Maintenant l'exécution de la fonction se termine par :

```
label pointretour ;
go-to adrretour ;
```

on rappelle que "adrretour" est une cellule du bloc mémoire local qui contient l'adresse programme à l'appel de la procédure où l'exécution doit reprendre.

Les instructions BL/1 données ci-dessus constituent une traduction complète de la procédure PL/1 de départ. Le programme qu'elles spécifient effectuent le même calcul que le texte PL/1 original et de la même manière. D'une manière informelle, ceci devrait rendre plus claire l'expression "équivalence sémantique" de cette thèse.

2.6 - Conclusions sur les langages de base pour PL/1 -

Le langage BL/1 présenté ici est un des candidats à une base de PL/1. Pour tout programme PL/1 de nombreux équivalents BL/1 sont possibles aussi. Par exemple, le test de la clause IF ci-dessus pourrait être traduit plus aisément par :

larger M, zéro → plus grand ;
not plusgrand → plusgrand ;

au lieu de :

larger zéro, M → pasplusgrand ;

qui est plus simple.

BL/1 est le résultat d'un effort tendant à définir une base PL/1 d'une manière ad hoc en regardant simplement ce qui était nécessaire pour toute instruction PL/1. De tels choix sont extrêmement arbitraires et devraient être revus à la lumière d'autres propriétés désirables du langage. L'essai par exemple de l'implémentation de D. SERAIN [21] conduit à une meilleure compréhension de cet aspect du problème et devrait être très utile pour toute révision de BL/1. On peut en dire autant du travail d'optimisation effectué actuellement par M. EXEL

L'élaboration d'un langage de base est évidemment un problème cousin de celle d'un langage quelconque : c'est un processus itératif. BL/1, comme on l'a présenté ici, est seulement la première itération de toute une série. La seconde phase de ces itérations est l'étude de toutes les traductions possibles de PL/1. L'orientation générale est donnée par le chapitre qui suit et les traductions actuelles considérées sont résumées en appendice D.

3 - LA TRADUCTION PL/1 - "BASE" -

3.0 - Introduction -

Le langage de base défini dans le chapitre précédent est la cible de la traduction de PL/1. Cette traduction qui doit être accomplie ultimement par macro-syntaxiques sera esquissée dans la première partie de ce chapitre (3.1) pour donner l'idée générale de la stratégie. Une spécification plus précise des points à traduire se trouvent en appendice - sous la forme de "transformations syntaxiques". PL/1 étant un langage immense beaucoup manquera dont les on-conditions, le traitement de multi-tâche et les entrées/sorties.

La seconde partie de ce chapitre (3.2) est consacrée à une discussion des problèmes que soulève la réalisation de transformations spécifiées par les macro-syntaxiques. On montrera en particulier pourquoi l'usage du langage préfixé (notation dite Polonaise) est utile comme étape intermédiaire de traduction plutôt que d'essayer de passer directement d'expressions PL/1 en leur équivalent BL/1. Puis pour contre balancer cette méthode fort classique, qu'elle est inapplicable pour les tableaux incomplètement spécifiés("cross-sections") et de là qu'on doit utiliser d'autres techniques. Ces deux exemples sont effectivement traités par le macro-analyseur et les résultats exhibés en appendice.

3.1 - La stratégie de la traduction -

3.11 - Partie déclarative de PL/1 -

Une caractéristique commune des compilateurs (voire même des assembleurs) est la disposition d'un outil pour rassembler l'information concernant les symboles utilisés dans le programme et utiliser cette information lorsqu'ils sont référencés.

Cette information déclarative en PL/1 n'inclut pas seulement les instructions DECLARE mais aussi les étiquettes en préfixes (LABEL) les noms de procédure et d'autres déclarations contextuelles et implicites.

3.12 - Représentation BL/1 des types de données PL/1 -

L'approche que l'on expose ici doit être considérée comme très typique de la philosophie générale de la traduction : à savoir que les données complexes sont réduites en combinaisons de types plus primitifs qui à leur tour le sont en types encore plus primitifs pour arriver à des déclarateurs uniquement BL/1.

Trois classes d'attributs de données sont reconnues :

a) - Types simples
b) - Types standardisés
c) - Types programmés

- Les types simples ont déjà un équivalent exact dans le langage de base.

  Par exemple POINTER est identique à une adresse en BL/1. Evidemment la traduction est ici seulement une manière de changer de notations.

- Les types standardisés doivent être représentés par une combinaison des types du langage de base mais pour lesquels néanmoins la correspondance est encore directe.

  Par exemple COMPLEX qui est une structure à deux types arithmétiques élémentaires.

- Les types programmés ont une représentation dont l'explicitation n'est pas réellement complète en elle-même : tout accès à leur valeur résulte de l'exécution d'un certain nombre d'actions.

  Par exemple : FILE, TASK.

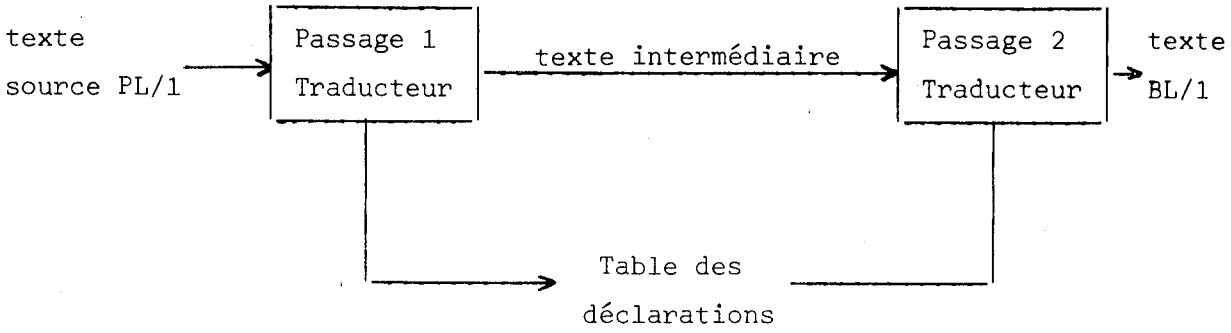
3.13 - Transmission des déclarations entre les passages -

Il est inconcevable qu'un PL/1 complet soit implanté en un seul passage puisque l'une de ses règles clefs est que les déclarations peuvent apparaître n'importe où dans leur bloc, même si elles apparaissent après qu'une référence à la déclaration ait été faite.

Par suite, une technique a été développée pour ranger l'information collectée au premier passage, information disponible par la suite.

Les fonctions qui peuvent apparaître dans la clause prédicative de la macro syntaxique accomplissent ces tâches.

On peut schématiser le système ainsi :



Le premier passage enregistre chaque déclaration explicite avec son identificateur, chaque sous-structure possible, valeur ou expression qui peuvent être utilisées quelque part dans le deuxième passage.

Par exemple, l'attribut INITIAL des variables contrôlées.
Le début et la fin de chaque bloc et de chaque bloc de PROCEDURE est noté de manière à associer identificateur et portée propre.

Chaque déclaration contextuelle ou implicite possible est enregistrée (décision prise à la fin de chaque bloc qui détermine la déclaration réelle).

Les DECLARE PL/1 sont éliminés du texte intermédiaire puisque leur fonction est remplacée par des références à la table de déclaration.

Le second passage utilise l'information collectée au passage précédent.

Toutes les déclarations qui apparaissent n'importe où dans un bloc particulier sont rassemblées en tête du bloc. En particulier les impli-

cites et contextuelles -maintenant complètement identifiées- apparaissent en tête du bloc de procédure externe (selon la définition du langage PL/1 lui-même).

On peut, à cette étape, appliquer l'attribut par défaut. Chaque expression appartenant à une déclaration particulière est alors analysée et traduite. Il est possible de retourner à la table de déclarations afin de l'insérer plus tard dans le texte.

Par exemple, c'est le cas des variables CONTROLLED et de l'instruction ALLOCATE.

Dans chaque cas, l'information déclarative complète est maintenant disponible dans la table des déclarations afin d'être accessible via une fonction-prédicat lorsqu'il est nécessaire d'obtenir l'information liée à l'identificateur.

Cette dernière prend normalement la forme d'une condition booléenne.

Par exemple : l'identificateur est-il FIXED BIN ? ou bien d'un sous-arbre syntaxique PL/1 (structure) ?

Par exemple : une expression analysée.

3.14 - Mise en oeuvre de l'information impérative -

On entend par information impérative, les instructions qui sont des commandes directes au calculateur pour accomplir une action. Les déclarations, elles, décrivent la manière d'interpréter une action ailleurs.

Les "impératifs" sont usuellement référencés comme des instructions actives du langage. L'application des macros est ici plus évidente que pour les déclarations puisque chaque définition de macro est en quelque sorte une manière de spécifier comment doivent être traduits en langage objet les appels à partir de leur définition. Ce seront des appels de macros directs ou par l'intermédiaire d'autres macros, ou des appels récursifs.

A ce point de vue les "impératifs" PL/1 peuvent être considérés

comme des abréviations des instructions du BL/1. Ici les macros-syntaxiques spécifient les relations exactes entre abréviations et expansions. Notons que ces expansions sont conditionnelles et sont fonction de l'information déclarative rassemblée au passage précédent.

Si le mécanisme de traduction que l'on expose ici reste unique pour tous les passages, il y a différentes manières de l'appliquer selon la nature des instructions PL/1 à traduire.

On appelle ces méthodes :

a) - "réduction directe"
b) - "réduction indirecte".

3.141 - Réduction directe -

On l'utilise dans les situations où l'instruction PL/1 donnée se réduit à un "petit" nombre d'instructions BL/1 qui peuvent être exécutées en une fois.

Pour illustrer la manière d'appliquer la méthode, considérons la traduction des instructions d'affectation. Les pas de cette traduction sont les suivants :

(1) Les affectations de structure et de tableaux sont réduites en affectations élémentaires :

Exemple :

```
...; DECLARE A (100) ;
     A = A + 7 ; ...
```

L'affectation du tableau est traduite ainsi :

```
...; DO K = 1 TO 100 ;
     A (K) = A (K) + 1 ;
     END ; ...
```

(2) Chaque expression "infixe" trouvée est traduite en une série d'affectations ayant chacune un seul opérateur

Exemple :

```
... ; X = B * B - 4.*  A * C ; ...
```

est traduite en

```
... ; t1 = B * B ; t2 = 4. * A ; t3 = t2 * C ;
X = t1 - t3 ; ...
```

(3) Si les opérandes d'un opérateur d'affectation ne sont pas d'un type primitif et ne correspondent pas au type demandé par la définition de l'opérateur, alors des conversions sont insérées.

Exemple :

```
... ; DECLARE C CPLX, A FIXED BIN ;
      C = C + A ; ...
```

cette dernière est traduite :

```
... ; t = COMPLEX (A, 0) ; C = C + T ; ...
```

qui à son tour devient :

```
... ; t.REAL = A ; t.IMAG = 0 ; C = C + t ; ...
```

Si tous les opérandes sont de type primitif alors les conversions sont faites en BL/1. On ne peut aller plus loin puisque la forme machine des primitives n'est pas spécifiée.

(4) Finalement, s'il reste des affectations non primitives, elles sont traduites en leur type primitif équivalent :

Continuons l'exemple précédent :

```
... ; C = C + t ; ...
```

est finalement traduit en :

```
... ; C.REAL = C.REAL + t.REAL ;
      C.IMAG = C.IMAG + t.IMAG ; ...
```

3.142 - Réduction indirecte -

Il est clair que la méthode qui vient d'être décrite ne fonctionnerait pas pour des traductions "volumineuses".

Le résultat final serait de grands blocs dont le code contiendrait beaucoup de répétitions.

Les programmeurs, bien sûr, ont rencontré ce genre de problèmes depuis longtemps et en ont donné une réponse classique : la création de sous-programmes.

Nous ferons souvent de même : le résultat de l'expansion de macros n'est pas le code qui sera exécuté mais plutôt un appel ou un ensemble d'appels à un sous-programme au moment de l'exécution (objet-time subroutine).

Un exemple classique de cette méthode de traduction sont les instructions GET et PUT.

Il suffit de considérer le nombre de conventions associées aux options LIST et DATA pour voir que si la programmation n'en est pas difficile, elle est plutôt compliquée :

Citons les différentes sortes de délimiteurs possibles, les conventions spéciales de fin de ligne, les conversions à introduire en fonction des différents supports d'entrée, la table des symboles exigée par le GET DATA.

Le FORMAT pose des problèmes spéciaux : certains seront traités comme du code destiné à un interpréteur, d'autres compilés en sous-programmes spéciaux ou en 'coroutines'.

3.2 - Réalisation en macro-syntaxiques -

L'une des activités, et la plus délicate, pour les usagers de traducteurs guidés par la syntaxe est de choisir une grammaire pour le langage particulier qu'ils considèrent, afin d'en obtenir le maximum d'efficacité. Bien souvent aussi, il s'agit de trouver une grammaire quelconque qui satisfasse aux contraintes du processeur !

Dans notre cas la situation est différente. D'abord l'algorithme que nous utilisons pour implémenter le macro-analyseur est "général" (cf. Early [9]). Le problème se pose beaucoup moins quant au choix de la grammaire qui satisfasse le problème d'analyse syntaxique, quoique l'écriture de macros-syntaxiques se pose dans les mêmes termes. La généralité de l'algorithme d'analyse syntaxique est pleinement justifié à priori, par conséquent : "on ne peut courir après deux lièvres à la fois". D'autre part, suivant les formes intermédiaires du langage, un ensemble de macros-syntaxiques peut être souvent radicalement réduit et clarifié. C'est l'objet de la démonstration présente : montrer par des exemples réels issus de mon travail sur PL/1 comment appliquer ces techniques.

3.21 - Conversion infixé vers BL/1 -

Commençons par un pas critique, et apparemment décevant, de la traduction d'expressions structurées en forme d'arbre (du langage origine c'est-à-dire PL/1) en une série d'instructions BL/1 qui exécutent une seule opération à la fois.

Par exemple la phrase PL/1 :

```
A = B * (C + D) * E ;
```

est, en BL/1 équivalent,

```
add C, D → t ;
mult B, t → t ;
mult t, E → A ;
```

On reconnaîtra ce problème aisément comme celui de la production d'un code intermédiaire. Tout "écrivain" de compilateur aura affronté ce type de problème. Il est donc utile d'examiner la solution par macros-syntaxiques. Il y a plusieurs techniques possibles pour effectuer cette traduction. L'une, appelée ci-dessous méthode directe, serait d'écrire des macros qui reconnaissent les expressions infixées comme des appels et produisent BL/1 directement dans leurs remplacements. L'autre, ou méthode indirecte, serait une traduction de la forme infixée en préfixée puis à l'aide d'un ensemble de macros de plus bas niveau, la traduction de préfixé en BL/1.

Si à priori, il n'y a pas de raison particulière de se méfier de l'une de ces méthodes, on constatera que la méthode directe est fastidieuse tandis que l'autre est naturelle. L'examen des deux méthodes conduira par comparaison à éclairer ce propos.

Pour l'une ou l'autre, il est nécessaire d'avoir une grammaire classique des expressions PL/1. Cette grammaire reflète à la fois l'associativité et la précédence des opérateurs.

```
expr → opd2 | expr '|' opd2
opd2 → opd3 | opd2 '&' opd3
opd3 → opd4 | opd3 rop opd4
opd4 → opd5 | opd5 '||' opd5
opd5 → opd6 | opd5 aop opd6
opd6 → opd7 | opd6 mop opd7
opd7 → opd8 | uop opd8 | opd8 '**' opd7
opd8 → '(' expr ')' | idf '(' lix ')' | idf | nombre
```

```
rop → '<' | '¬<' | '<=' | '=' |
      '¬=' | '>=' | '>' | '¬>'
aop → '+' | '—'
mop → '*' | '/'
uop → '+' | '—' | '¬'
```

Noter que l'associativité est à gauche sauf pour <opd7> qui spécifie l'associativité à droite des opérateurs d'exponentiation et des opérateurs unaires. La longue liste de non-terminaux (<expr>, <opd2> ... <opd8>) est la méthode de spécification syntaxique de la précédence des opérateurs PL/1.

3.211 - Une méthode directe -

Maintenant considèrons comment écrire les macros-syntaxiques qui accepteront les expressions écrites suivant la syntaxe ci-dessus et qui les analyseront comme si elles étaient écrites en leur équivalent BL/1.

Comme de coutume, on trouvera la solution au problème assez facilement en les considérant en termes de transformations syntaxiques.

Reprenons l'expression PL/1 :

A = B * (C + D) * E ;

On doit trouver la dernière opération BL/1 à appliquer et transformer l'expression afin de placer cette opération dans la chaîne d'entrée :

t = B * (C + D) ; mult t, E → A ;

A noter qu'un nouvel identificateur t devra être créé pour contenir le résultat de la première sous-expression. Appliquant la même règle de transformation à la nouvelle affectation, on devra produire :

t2 = (C + D) ; mult B, t2 → t ;

Finalement, supprimant les parenthèses et reconnaissant la dernière affectation, l'équivalent BL/1 à produire devra être :

add C, D → t2

Est-il clair que ce processus est équivalent à la sélection et à l'élimination progressive d'une expression de l'arbre syntaxique en commençant par l'opérateur le plus haut et en le "réécrivant" à sa droite ? En théorie, il devrait être possible de choisir le plus bas et de le "réécrire" à sa gauche. On envisagera les complications soulevées en essayant de trouver cette opération dans l'arbre !

En fait le procédé est déjà sérieusement désavantageux comme on peut le voir en essayant d'écrire une macro à appliquer à la transformation envisagée. L'appel est clairement au niveau de l'affectation :

stat → idf_g '=' expr ';'

(<idf>$_g$ est le seul identificateur permis à gauche de l'affectation ; l'exemple est simplifié pour plus de clarté.

L'expression doit être examinée pour déterminer l'opérateur le plus haut. Une cascade de prédicats de sous-structure effectue cette traduction "dans le style" :

```
stat → idf_r '=' expr ';'
  = si expr →* expr_g '|' opd2_d
    alors ...
    sinsi expr →* opd2
    alorsi opd2 →* opd2_g '&' opd3_d
    alors ...
    sinsi opd2 →* opd3
    alorsi opd3 →* opd3_g rop opd4_d
    alors ...
    sinsi ...
```

(Note pour la compréhension de la notation : les chiffres (1), (2) ... servent de repères pour discussion, les indices g, d, t, r indiquent gauche, droite, temporaire, résultat et la flèche $\overset{*}{\rightarrow}$ ... indiquent "sous-structure profonde" c'est-à-dire aller jusqu'à trouver ...).

Dans la macro au-dessus, chaque paire d'une alternative commençant à <expr> est examinée à son tour jusqu'à trouver celle qui contient un opérateur. Supposons que cela se produit pour l'opérateur plus (+). Le prédicat correspondant se développera ainsi :

```
          ...
(1)       alorsi opd5 → opd5g opa opd6d
(2)       alors si opa → '+'
(3)            alors si opd5g *→ idfg
(4)                 alors si opd6d *→ idfd
(5)                      alors 'add' idfg ',' idfd '→' idfr ';'
(6)                      sinsi idft : nouveau
(7)                      alors idft '=' opd6d ';'
(8)                           'add' idfg ',' idft ' ' idfr ';'
(9)                      fsi
(10)                sinsi idft : nouveau
(11)                alors idft '=' opd5g ';'
(12)                     idfr '=' idft '+' opd6d ';'
(13)                fsi
(14)           sinsi opa → '-'
               ...
```

Les prédicats (1) et (2) établissent que "plus" est trouvé. Dans ce cas les prédicats (3) et (4) déterminent si les parties gauche et droite sont des identificateurs simples. Si oui, le remplacement (5) est sélecté : il convertit l'affectation directement en sa forme BL/1. C'est la traduction de :

```
C = A + B ;
```

qui satisfait ces prédicats pour produire le remplacement

```
add A, B → C ;
```

Maintenant, si la partie gauche est un identificateur simple mais non la partie droite (le prédicat (4) échoue) alors un identificateur unique (<idf>$_t$) est créé par la fonction-prédicat "nouveau" (6). Cet identificateur contiendra le résultat de l'évaluation de la partie droite (7) et sa valeur sera ajoutée par une instruction BL/1 à l'identificateur de la partie gauche (8) :

```
D = A + B ** C ;
```

devient

```
t = B ** C ;
add A, t → D ;
```

A noter que la nouvelle affectation créée par le remplacement à la ligne (7) est bien sûr l'objet d'un nouvel appel de l'entière macro.

Enfin, si le prédicat (3) échoue c'est que l'opérande gauche de la somme n'est pas un simple identificateur. Deux affectations sont créées par (11) et (12). La première place le résultat de la partie gauche dans une variable temporaire créée en (10) et la seconde substitue celle-ci à la place de celle-là. A noter un nouvel appel de notre macro à la fin de (12) qui, elle, testera la partie droite (puisque (3) sera satisfait).

```
E = A/B + C * D ;
```

devient

```
t = A/B ;
E = t + C * D ;
```

Ecrire la macro tout entière pour décomposer les expressions PL/1 en langage de base serait extrêmement long (à tous points de vues). Le développement de l'opérateur plus longuement détaillé ici serait à répéter d'une manière analogue pour tout le "répertoire" des opérateurs de PL/1 (des améliorations sont possibles mais les répétitions sont inévitables).

Il est instructif d'examiner la raison de cette "débacle". La cause essentielle en est la multitude de prédicats associés au grand nombre de non-terminaux. Dans toute analyse syntaxique, il y a de nombreux non-terminaux (en correspondance un-un) dont la seule fonction est d'établir la liaison des opérateurs. C'est le résultat classique de l'utilisation de grammaires hors-contexte pour établir la précédence des opérateurs. En dépit du fait que ces non-terminaux "pont" ne sont pas réellement utiles, ils doivent être examinés par les prédicats pour cheminer dans l'arbre ("en descendant").

3.212 - Une méthode indirecte -

Si le nombre de non-terminaux non essentiels est la source du problème, on devrait trouver une grammaire pour les expressions qui en contienne moins. Si les expressions étaient représentées en notation préfixée (ou infixée) la plupart des non-terminaux disparaitraient puisqu'il ne serait plus besoin de spécifier les précédences.

La grammaire préfixée serait :

opérande → identificateur
| opérateur-unaire opérande
| opérateur-binaire opérande_1 opérande_2 | ...

3.2121 - PREFIXE VERS BL/1 -

Les transformations explicitées en 3.211 s'appliquent également aux expressions infixées, pré ou postfixées. Par exemple pour la notation préfixée il y a les trois mêmes transformations :

	(infixée)	(préfixée)	(préfixé transformé)
(a)	C = A + B ;	= C + AB ;	add A, B → C ;
(b)	D = A + B ** C ;	= D + A ** BC ;	= t ** BC ; add A, t → D ;
(c)	E = A/B + C * D ;	= E + /AB * CD ;	= t /AB ; E + t * CD ;

Lorsque ces transformations sont appliquées à chaque affectation préfixée d'une manière répétitive, il est clair que BL/1 seul restera.

Elles peuvent être implémentées par une macro syntaxique qui rappelle fortement une partie de celle décrite pour les expressions infixées :

```
(1)  instr → '=' idf_r opd_s ';'
(2)  = si opd_s → opb opd_g opd_r
(3)    alors si opd_g → idf_g
(4)          alors si opd_d → idf_d
(5)                alors si opb → '+' alors 'add' idf_g ',' idf_d '→' idf_r ';'
(6)                      sinsi opb → '—' alors 'subtr' idf_g ',' idf_d '→' idf_r ';'
...                      sinsi ...
(7)                      fsi
(8)                sinsi idf_t : nouveau
(9)                alors '=' idf_t opd_d ';'
(10)                     '=' idf_r opb idf_g idf_t ';'
(11)               fsi
(12)         sinsi idf_t : nouveau
(13)         alors '=' idf_t opd_g ';'
(14)               '=' idf_r bop idf_t opd_d ';'
(15)         fsi
(16)   sinsi opd_s → uop opd
...    ...
(17)   fsi
```

La transformation (a) est effectuée de (5) à (7), (b) de (9) à (10) et (c) de (13) à (14).

3.2122 - INFIXE VERS PREFIXE -

Discuter des grammaires préfixées peut sembler retourner à la question de savoir quoi faire de la grande taille de la macro infixée. Après tout, le texte source est nécessairement en notation infixée. Il s'avère en fait qu'il existe une méthode aisée de transformation d'infixée

en préfixée. Le volume total de la méthode à deux pas (infixée vers préfixée, préfixée vers BL/1) est bien moins important que la méthode directe donnée au début.

La manière la plus simple de traduire d'infixée en préfixée est aussi indirecte dans le sens que la "source" est d'abord complètement analysée par la grammaire infixée. On considère celle-ci comme un ensemble de règles de transitions (macros sans remplacement). Le premier appel effectif de macro est au niveau de l'instruction d'affectation :

```
inst → idf '=' expr ';'
   = '=' idf 't' expr ';'
```

A noter que le remplacement serait en préfixée si la phrase

't' expr

était un opérande. Ce soi-disant opérande va l'être tout à fait par la macro :

```
opérande → 't' expr
= si expr → expr_g '|' opd2_d
  alors '|' 't' expr_g 't2' opd2_d
  sinsi expr → opd2
  alors 't2' opd2
  fsi
```

A noter encore que l'on trouverait une forme préfixée si les phrases

't' expr et 't2' opd2

étaient des opérandes. La première est prise en compte par la même macro tandis que la seconde est traitée par une macro de la même forme mais adaptée au second niveau pour la grammaire infixée. Les marqueurs t, t2, t3, ... sont transmis en descendant l'arbre infixé qui traduit vers le préfixé ce faisant. Les marqueurs changent d'identité à chaque niveau de façon à éviter des ambiguïtés qui sans cela seraient inévitables.

Qu'arrive-t-il au dernier niveau Comment les marqueurs sont-ils éliminés ? On représente ici ce niveau par :

opérande → 't8' opd8

= si opd8 → idf alors idf

sinsi opd8 → '(' expr ')' alors 't' expr

sinsi ...

Le marqueur est simplement effacé dans le cas d'un identificateur simple puisqu'un identificateur est un opérande de préfixé. Pour une expression parenthésée les parenthèses sont enlevées et tout le processus repart au premier niveau par le marqueur t. Finalement l'arbre d'expression entier sera sous forme préfixée.

3.213 - Comparaison -

A-t-on réduit raisonnablement le nombre de non-terminaux dont il était question au départ, afin de simplifier l'ensemble des macros à écrire ? Après tout, tous les non-terminaux de la grammaire infixée sont encore présents dans la solution à deux étapes ! Clarifions la situation : ces prédicats s'appliquant à la grammaire infixée ont une ligne simple de remplacements au lieu d'un développement redondant dans chaque cas. De plus la méthode en deux étapes permet la factorisation des transformations redondantes à une seule sous-section.

Une observation finale sur cet exemple est qu'il met en lumière le phénomène important de décomposition en couche des macros-définitions. Cette exemple révèle le phénomène important de "couches" de langages dans les définitions de macros. Une expansion de macros en un ensemble d'appels de macros correspond à la suppression d'une couche du langage source. L'expansion des macros du niveau inférieur ôte la seconde couche et ainsi de suite jusqu'au texte de base. Si les expansions de toutes les couches progressent concurremment en scrutant le texte, le travail pour chaque couche est logiquement indépendant de celui de toutes les autres couches. La seule condition préalable est que les "langages de sortie" (expansions) d'un niveau soient les mêmes que ceux du "langage

d'entrée" (calls) du niveau immédiatement inférieur. Les transformations linguistiques qui interprètent l'infixé comme code préfixé sont absolument indépendantes de celles qui interprètent le préfixé comme du code BL/1. Cela signifie qu'on pourrait aisément outrepasser la couche et donner le code préfixé comme source au macro-analyseur sans différence décelable. Réciproquement, on pourrait décider arbitrairement de traiter le préfixé comme langage de base. Le code source -écrit en infixé- serait analysé comme préfixé et serait directement généré à partir de cette dernière forme.

Remarque :

Ceci rappelle fortement l'une des idées de P. POOLE pour l'implémentation d'une "machine abstraite". Les processeurs sont écrits comme des cascades de macros. Il suggère d'améliorer l'efficacité du système - si on le désire - en implémentant les macros de plus haut niveau - et celles-là seulement - au lieu et place d'appel à des macros de plus bas niveau [19].

Les macros décrites dans cette partie sont bien sûr un exemple réduit choisi parmi les techniques de traduction disponibles. Il y a en fait bien plus de "couches" entre PL/1 et BL/1. Cet exemple correspond réellement au second niveau de la réduction directe décrite en 3.14 : c'est dire que le langage de sortie ne devrait pas être BL/1 mais une forme d'opérateurs simples dont le traitement en couches de niveaux inférieurs prendraient en compte les conversions de données et des données non primitives avant la production effective de BL/1.

3.22 - Tranches de tableaux ("cross-sections") -

C'est à la traduction du premier niveau de PL/1 que les plus sérieux problèmes de macros-syntaxiques sont souvelés. Il est nécessaire ici de traiter directement l'expression infixée. Ce paragraphe traite des nouvelles méthodes à appliquer et montre à l'aide d'un exemple assez difficile que si les transformations syntaxiques sont aisément imaginables, les macros ne sont pas nécessairement écrites directement.

Cet exemple décrit ici est le traitement des tranches de tableaux (cross sections) en PL/1. Prenons par exemple une instruction du type :

```
A(1, *) = B(*, K) + C(J, *) + D ;
```

on la compilera comme

```
DO i = LBOUND(A, 2) TO HBOUND (A, 2) ;
  A(1, i) = B(i, K) + C(J, i) + D ;
  END ;
```

Exprimons verbalement les différents pas de la transformation :

(a) la liste d'index en partie gauche de l'affectation est examinée de gauche à droite jusqu'à trouver le premier astérique (*).

(b) la position de l'astérique détermine la dimension du tableau à spécifier pour déterminer les limites de l'itération.

(c) une nouvelle variable est créée et une instruction DO émise qui "bouclera" pour la variable d'itération entre les bornes de la dimension spécifiée.

(d) chaque référence au tableau est examinée dans l'affectation de gauche à droite : la variable d'itération remplace le premier astérique trouvé. L'affectation modifiée est placée après l'instruction DO.

(e) une instruction END est émise pour clore l'itération.

Il devrait être clair que la transformation s'applique répététivement. Pour l'instruction :

```
A(*, 1, *) = -B(*) + C(2, 3, *, *) ;
```

la première application de la transformation conduit à :

```
DO i = LBOUND(A, 1) to HBOUND(A, 1) ;
  A(i, 1, *) = -B(i) + C(2, 3, i, *) ;
END ;
```

et la seconde à :

```
DO i = LBOUND(A, 1) to HBOUND(A, 1) ;
   DO j = LBOUND(A, 3) to HBOUND(A, 3) ;
     A(i, 1, j) = -B(i) + C(2, 3, i, j) ;
   END ;
END ;
```

Comment exécuter ces transformations en macros-syntaxiques ? Tout d'abord effectuer la "recherche du premier indice non spécifié du tableau à gauche de l'affectation et remplacer cet indice par une variable d'itération" doit l'être par les outils élémentaires dont on dispose jusqu'ici (la fonction-Prédicat "ad hoc" n'existe pas). La difficulté provient de l'affectation infixée elle-même puisque les changements ne sont pas au niveau de l'appel de la macro.

On considère le premier pas cité ci-dessus en comptant le nombre de fois où un marqueur spécial appelé scruter franchira un indice spécifié à partir de la gauche de la liste d'indice du tableau :

```
instr → destination '=' expr ';'
destination → idf | idf '(' lix ')'
lix → ix | ix ',' lix
ix → expr | '*'
expr   ... (voir plus haut)
```

le marqueur scruter est implanté en partie gauche de la liste d'index (<lix>) par la macro :

```
(1)  instr → destination '=' expr ';'
(2)      =  si destination → idf '(' lix ')'
(3)         alors idf '(' 'scruter' '0' lix ')=' expr ';'
(4)         sinsi destination → idf
(5)         alors '2' destination '=' expr ';'
(6)         fsi
```

Comme il est d'usage, <destination> a une alternative considérée en (2) et (4). Si la partie gauche est un identificateur simple (prédicat (4)) alors aucune transformation ne s'applique et l'affectation est passée au second niveau du langage (en utilisant le marqueur 2 pour éviter une boucle d'appels). Une référence à un tableau (prédicat (2) vrai) détectée, le marqueur scan suivi du nombre zéro (0) indique le début de la liste. La valeur associée à la catégorie <nombre> est incrémentée en appliquant le prédicat fonction "incr" avec l'arguement <nombre> :

incr (nombre)

Marqueur et nombre insérés dans la liste est la base avec <lix> de la prochaine macro détectant l'astérisque :

```
(1)  lix → 'scruter' nombre lix2
(2)      = si incr(nombre)
(3)        alorsi lix2 → ix3 ',' lix3
(4)        alors si ix3 → '*'
(5)              alors si idfi : niveau
(6)                   alors idfi ',' lix3 'trouvé' nombre idfi
(7)                   fsi
(8)              sinon ix3 ',' |'scruter' nombre lix3|
(9)              fsi
(10)       sinsi lix2 → ix3
(11)       alors si ix3 → '*'
(12)             alors si idfi : nouveau
(13)                  alors 'trouvé' nombre idfi
(14)                  fsi
(15)             sinon lix2 'pastrouvé'
(16)             fsi
(17)       fsi
```

<nombre> est incrémenté par le prédicat (2) puis la liste d'index <lix> est examinée par le prédicat (3) pour déterminer le nombre de composants (supérieur à 2). Dans ce cas le prédicat (5) crée un nouvel identificateur lorsque l'index est '*' pour le substituer. La sous-phrase au début de la ligne (6) :

$$idf_i \text{ ',' } lix_3$$

sera reconnue comme un cas courant de <lix>. Le reste de ce remplacement (5) est précisé par le marqueur trouvé suivi de son nombre de dimension (<nombre>) et le nom de la variable d'itération trouvée ($<idf>_i$).

Lorsque l'index précédent n'est pas astérisque ((4) échoue) alors le marqueur scruter est déplacé de un vers la droite et la macro <lix> est réappelée pour la portion soulignée du remplacement (8).

La liste d'indices peut être réduite à un seul index (prédicat (10)). Alors c'est soit la même action effectuée ci-dessus (remplacement (13)) ou il n'y a aucun astérique dans la liste. C'est indiqué par le marqueur pastrouvé dans le remplacement (15).

S'il n'y a pas d'astérisque cela signifie qu'aucun traitement pour spécification incomplète d'indice n'est nécessaire et l'affectation est passée au niveau du second langage :

```
instr → idf '(' ixl 'pastrouvé' ')=' expr ';'
    = '2' idf '(' ixl ')=' expr ';'
```

D'autre part, la création de la boucle DO pour un index non spécifié s'effectue en remplaçant l'astérisque le plus à gauche pour chaque référence de tableau à droite de l'affectation :

```
instr → idf_t '(' lix 'trouvé' nombre idf_i ')=' expr ';'
    = 'DO' idf_i '=LBOUND(' idf_t ',' nombre ')'
        'TO HBOUND(' idf_t ',' nombre ');'
        idf_t '(' lix ')=' | 'remplacer' idf_i expr | ';'
      'END;'
```

Ici, la technique de remplacement des identificateurs de variables d'itération est analogue à celle de la conversion d'infixée vers le préfixée : le marqueur est transmis à travers les branches de l'arbre syntaxique en descendant jusqu'à trouver des références de tableau. Les marqueurs sont alors transmis de gauche à droite dans les listes d'indices jusqu'à trouver '*' qui sera remplacé par le nom de la variable d'itération transportée tout du long.

La phrase encadrée (□) ci-dessus est semblable à une expression Ainsi une macro <expr> qui transmet le marqueur remplacer et le nom d'une variable d'itération au niveau syntaxique inférieur peut être :

```
expr → 'remplacer' idf_i expr_2
   = si expr_2 → expr_g '|' opd2_d
     alors 'remplacer' idf_i expr_g '|'
           'r2' idf_i opd2_d
     sinsi expr_2 → opd2
     alors 'r2' idf_i opd2
     fsi
```

Des macros similaires sont nécessaires à chaque niveau de la grammaire infixée jusqu'au niveau du primaire <opd8>

```
opd8 → 'r8' idf_i opd8_2
   = si opd8_2 → '(' expr ')' alors '(remplacer' idf_i expr')'
     sinsi opd8_2 → idf_e alors idf_e
     sinsi opd8_2 → nombre alors nombre
     sinsi opd8_2 → idf_t '(' lix ')' alors idf_t '(rix'
                                            idf_i lix')'
     fsi
```

Le but est atteint finalement avec la macro <lix> :

```
lix → 'rix' idf_i lix_2
    = si lix_2 → ix_g ',' lix_d
      alors si ix_g → '*'
            alors idf_i ',' lix_d
            sinon ix_g ', rix' idf_i lix_d
            fsi
      sinsi lix_2 → ix
      alors si ix → '*' alors idf_i sinon lix_2 fsi
      fsi
```

3.23 - Sommaire -

Il n'a pas été tenté, dans ce qui précède, d'obtenir des macros-syntaxiques complètes pour la traduction de PL/1 puisque l'effort a porté directement sur la découverte de quelques problèmes liés à cette traduction. Il ne fait aucun doute que ces exemples laissent une impression de grande complexité. Cette complexité est due pour une bonne part à la nature inhérente des grammaires qu'on doit traiter plutôt qu'au mécanisme des macros-syntaxiques lui-même. Des exemples d'opérations plus élaborées pour une grammaire préfixée d'Algol 68 seront présentés dans les chapitres qui suivent. Ces exemples contrairement à PL/1 sont pourtant assez directement écrits.

Le macro-analyseur peut être néanmoins nettement amélioré surtout pour les fonctions-Prédicats. A ce jour, nous leur avons délibérément conservé une forme primitive afin d'obtenir par des essais -fonctionnels ou descriptifs- ce qui est réellement fondamental et nécessaire dans un tel système. Si ces fonctions étaient étendues et plus puissantes on contribuerait à réduire le nombre de remplacements de "style LISP" pour manipuler l'arbre syntaxique. Le dernier exemple 3.22 en est une bonne illustration.

4 - LES ARBRES SEGMENTES COMME CODE INTERMEDIAIRE -

Jusqu'ici on a accordé beaucoup d'attention aux propriétés linguistiques du langage de sortie défini par expansions de macros. Mais dans la pratique, on doit s'occuper aussi de la forme interne de cette information. Ce chapitre sera consacré à ce problème en mettant en lumière certains des résultats de mon travail sur le compilateur ALGOL 68 grenoblois.

On utilise très souvent les arbres d'opérateurs comme code intermédiaire d'un compilateur. Le désavantage principal de cette méthode est l'espace mémoire occupé par l'arbre pour un grand programme.

(La réalisation physique d'un arbre utilise typiquement les mêmes éléments et dans le même ordre que le code polonais postfixé. La seule exception est que chaque sous-noeud est pointé à partir de son opérateur au lieu d'être défini implicitement par position. Tous ces pointeurs grèvent l'occupation de l'espace).

Une solution typique du problème de l'espace est de limiter la construction de l'arbre à une seule instruction à la fois. Le résultat est qu'un minimum de l'espace est occupé mais on a quand même la souplesse donnée par les arbres parce que l'on peut choisir l'ordre de génération. (On n'est pas obligé de traiter les sous-éléments d'un opérateur, de gauche à droite).

4.1 - Segmentation en Algol 68 -

A première vue, il semble que l'on ne puisse pas appliquer cette méthode à ALGOL 68 parce que l'instruction est définie de telle façon qu'elle peut être indéfiniment longue [23]. En voici un exemple :

```
(1)        proc p = ent :
(2)            début
(3)                i1 ;
(4)  e :           v1 := (i2 ; i3 ; v2) ;
(5)                v3
(6)            fin ;
(7)        i4 ;
(8)        i5 ; ...
```

L'instruction se trouvant à l'étiquette e contient comme sous-instructions les instructions i2 et i3 et, en plus, l'affectation v1 := v2. Cette instruction est elle-même une sous-instruction de l'instruction qui est la procédure de lignes (1) à (6). Donc, à cause, de l'imbrication des instructions dans d'autres instructions, on pourrait penser qu'il n'y a aucune méthode satisfaisante pour fragmenter le texte d'un programme en arbres de taille convenable.

Un tel algorithme existe pourtant, mais pour l'expliquer clairement, il faut regarder de plus près les problèmes particuliers d'ALGOL 68. Dans ce cas, le point difficile est la détermination des "modifications" dans un texte ALGOL 68. Dans la plupart des langages, les conversions de données sont un phénomène tout à fait local : elles sont définies syntaxiquement pour certains emplacements dans le texte. Comme, par exemple, au symbole d'affectation, à l'argument d'un appel, etc.... La production de code pour les conversions peut être entremêlée avec les autres générations sans grands problèmes. En ALGOL 68 la position syntaxique qui oblige telle ou telle modification peut être arbitrairement éloignée de la position où cette modification est appliquée. Le pire, c'est que le point où la modification est déterminée peut apparaître après l'endroit où le code aurait dû être produit. Prenons comme exemple :

a := si b alors c sinon d fsi + e

(La nature de l'opérateur plus (+) n'est pas déterminée jusqu'au moment où les modes des deux opérandes sont connus. Donc on doit lire jusqu'à e pour savoir quelles modifications appliquer à c et à d).

C'est pour cette raison que beaucoup d'écrivains de compilateurs ALGOL 68 ont décidé de séparer la détermination des modifications et la production de code et de les placer dans deux passages différents, bien que ceci pose des problèmes d'organisation supplémentaires.

Mais il est possible de déterminer un groupe de modifications sur un arbre d'opérateurs qui ne couvre pas tout le texte du programme. En effet la taille moyenne d'un fragment d'arbre peut être très limitée si l'on observe la règle très simple suivante :

" *Le mode à posteriori d'un segment de texte qui est terminé par un point virgule (;) est toujours neutre. Le mode neutre ne peut être modifié en aucun autre mode dans cette position ("hissage" est interdit). Donc ce segment peut être enlevé sans changer en rien le problème de la détermination pour le texte entourant".*

Voici une clef qui permet de fragmenter complètement un programme ALGOL 68, avec chaque fragment présentant un problème de détermination des modifications, problème indépendant de tous les autres. On procède ainsi : le texte original est découpé en plusieurs niveaux. Si une phrase contient une suite de phrases, chacune terminée par un point virgule, la suite est remplacée par un indicateur de descente (↓). La suite est mise au niveau inférieur et sa fin est délimitée par un indicateur de remontée (↑). Ce traitement est à la fois itératif (il s'applique aux autres suites de même niveau) et récursif (les phrases enlevées peuvent elles-mêmes contenir des suites de phrases).

4.11 - Un exemple de texte segmenté -

L'algorithme de segmentation sera appliqué au texte suivant :

```
début
    i1 ;
    a := (i2 ; i3 ; b • i4 ; i5 ; c)
         + (d | e := (i6 ; i7 ; f) ; i8 ; g | h) ;
    i9
fin
```

Le texte traité se divise en quatre niveaux :

niveau(1) : début ↑ i9 fin

niveau(2) : i1 ; a := (↓ b • ↓ c) + (d | ↓ g h) ; ↑

niveau(3) : i2 ; i3 ; ↑ i4 ; i5 ; ↑ e := (↓ f) ; i8 ; ↑

niveau(4) : i6 ; i7 ; ↑

Bien qu'il n'y ait pas de limite théorique pour le nombre de niveaux produits, il est très limité dans la pratique. D'une façon indicative, quand il y a un point-virgule à l'intérieur d'une paire de parenthèses, un nouveau niveau est créé. Pour avoir encore un autre niveau, cette même configuration doit se reproduire à gauche du point-virgule, etc.... Il est rare de trouver un programme avec plus de trois ou quatre niveaux.

Pendant le traitement (génération de code) du texte segmenté, il suffit d'avoir un compteur d'emplacements pour chaque niveau. Les limitations de cette méthode seront discutées plus tard (Section 4.13).

4.12 - L'utilisation du texte segmenté -

Chaque phrase du texte segmenté, c'est-à-dire celle qui est bornée par les points-virgules, est soumise à trois phrases de traitement :

(1) construction de l'arbre pour cette phrase

(2) détermination des modifications pour l'arbre créé

(3) production de code pour cet arbre.

Les deux premières phases se déroulent d'une façon autonome sans interruption. La troisième phase, la production du code, doit être suspendue quand un noeud indiquant une descente (↓) est rencontré. A ce moment, les mêmes trois phases doivent être appliquées successivement à chaque phrase se trouvant au niveau inférieur jusqu'à un indicateur de remontée (↑). Ensuite la phase originale de production de code est relancée. Cette action peut se reproduire plusieurs fois sur le même niveau et, bien entendu, récursivement sur les niveaux en-dessous.

4.13 - Les avantages et les inconvénients de la méthode -

L'objectif de l'arbre segmenté est d'exploiter la souplesse d'un arbre d'opérateurs sans utilisation excessive de la mémoire principale. L'idéal, qui est de traiter une seule instruction à la fois, n'est pas atteint par cette méthode. Par contre, on peut garantir qu'il existe une limite sur le nombre d'arbres qui doivent coexister à un instant donné pendant la génération. Ce nombre est le nombre des niveaux pour un programme donné. Comme déjà indiqué, il n'y a normalement que très peu de niveaux. Un cas très défavorable serait, par exemple, un arbre avec 30 noeuds (il faut rappeler que tous les éléments terminés par point-virgule ont déjà été enlevés). Si on suppose qu'il existe cinq arbres, un pour chaque niveau d'imbrication, l'espace maximum requis pour tout le programme serait celui nécessaire pour 150 noeuds. Il existe très peu de gens ayant la patience d'écrire un tel exemple ! Un exemple plus réaliste serait deux niveaux avec dix noeuds chacuns.

Si on regarde l'exemple de segmentation ci-dessus, il doit être clair que le texte pour cette portion du compilateur peut être considéré comme s'il avait été éclaté en plusieurs chaînes d'entrées indépendantes correspondant aux niveaux de segmentation. De la même manière, il y a un pointeur vers la position courante à chaque niveau. Comme conséquence on n'a pas besoin, pour l'espace mémoire, de plus d'une zone tampon pour chaque chaîne d'entrée.

Mais c'est précisément cet aspect de "chaînes d'entrée parallèles" qui pose un problème potentiel. Rappelons que la raison pour laquelle l'arbre d'opérateurs est plus convenable que l'organisation postfixée est que l'arbre permet à l'écrivain de compilateur de choisir l'ordre dans lequel on traite les opérandes. Mais, si un arbre particulier contient deux noeuds de descente (φ), il est nécessaire que la première suite de phrases du niveau en-dessous soit utilisée avant la deuxième. On pourrait penser que cette contrainte oblige à générer de gauche à droite et donc la négation de l'avantage poursuivi. On verra par la suite comment on peut éviter le problème, mais, d'abord, un exemple sera donné pour bien clarifier la situation.

L'instruction :

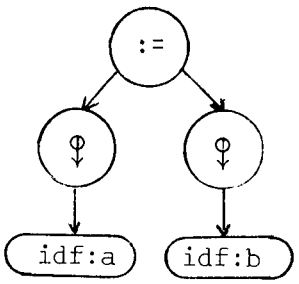
(i1 ; i2 ; a) := (i3 ; i4 ; b)

se divise en deux niveaux :

niveau(1) : (↓ a) := (↓ b)

niveau(2) : i1 ; i2 ; ↑ i3 ; i4 ; ↑

L'arbre d'opérateurs pour l'instruction sur le niveau(1) est :



Dans la plupart des langages on veut générer la partie droite d'une affectation avant la partie gauche pour minimiser le nombre de registres centraux utilisés. Si on procédait ainsi pour l'exemple donné le résultat serait de générer

i1 ; i2 ; ↑

à la place de

i3 ; i4 ; ↑

ce qui serait évidemment une erreur grossière.

Au lieu d'imposer simplement une génération de gauche à droite, il existe deux autres solutions à ce problème :

(1) - On peut modifier le format du noeud descente avec un pointeur explicite vers la suite des phrases enlevées. Le problème de trouver la bonne suite est résoluable, mais l'organisation des entrées-sorties pour le compilateur est sérieusement alourdie si l'on veut garder le texte en mémoire secondaire.

(2) - Pendant la construction de l'arbre on peut indiquer pour chaque noeud s'il est au-dessus d'un noeud descente. Ensuite, pendant la production de code, si l'on désire traiter un noeud hors de l'ordre canonique il suffit de regarder tous ses frères ainés non-traités. Si aucun ne couvre un noeud descente, l'action est permise. Bien entendu le résultat est que certaines occasions d'optimisation doivent être sacrifiées. Elles sont moins nombreuses que ce que l'on serait porté à croire. On voit pourquoi, si on considère qu'une technique classique pour minimiser le nombre de registres centraux occupés est de, toujours, générer le sous-noeud le plus "lourd" d'abord. Or, la présence même d'un noeud descente laisse supposer que le sous-noeud en question est "lourd" et donc doit être traité d'abord. C'est effectivement le cas dans l'exemple ci-dessus.

4.2 - La forme de l'arbre des opérateurs -

L'arbre des opérateurs est la base de la méthode proposée pour la génération de code d'ALGOL 68. Chaque noeud de cet arbre est composé :

(1) - d'un numéro d'opérateur indiquant les routines de détermination des modifications et de génération de code à appliquer,

(2) - d'un nombre fixe de pointeurs vers ses sous-noeuds (les opérandes) et,

(3) - d'un nombre fixe de données supplémentaires (les paramètres).

Les détails d'un algorithme qui sert à déterminer les modifications seront discutés dans le prochain chapitre. La génération de code proprement dit sera présentée dans le chapitre suivant.

Les algorithmes dépendent tous les deux de la forme de l'arbre décrit ci-dessus. La détermination des modifications sera décrite comme un processus qui "insère" des noeuds-opérateurs supplémentaires représentant ces modifications dans l'arbre. Pour notre compilateur, on préfère utiliser dans la pratique, une liste de modifications comme "paramètre" du noeud modifié. Ceci n'est qu'une autre réalisation physique de la même procédure logique. On utilise les listes pour éviter les problèmes intrinsèques de

modification d'un arbre qui sert simultanément de structure de contrôle pour l'algorithme.

5 - DETERMINATION DES MODIFICATIONS EN ALGOL 68 -

Dans le chapitre précédent nous avons vu comment un texte ALGOL 68 peut être découpé en petits segments relativement indépendants. On peut alors traiter chaque segment en deux phases :

(1) - détection des conversions de données ("modifications" en ALGOL 68) et

(2) - génération de code.

L'algorithme pour déterminer les modifications sur un arbre sera exposé dans ce chapitre.

A l'origine, j'ai écrit cet algorithme en ALGOL 68 lui-même, mais dans l'exposé ci-après il sera présenté au moyen de macros-syntaxiques, afin d'explorer le côté définition de celles-ci.

Les modifications sont un aspect assez délicat d'ALGOL 68. Pour cette raison, avant d'aborder l'algorithme en section 5.2, la première section (5.1) sera une explication des notions de "force du contexte" et de "classe de modification". La motivation des divers choix sera exposée.

5.01 - La version d'ALGOL 68 utilisée -

Malheureusement une discussion d'ALGOL 68 est toujours compliquée par la nécessité de spécifier quel langage est en question. La version officielle a été approuvée par un congrès IFIP, fin 1968. Depuis on a proposé beaucoup de changements mineurs (certains sont mêmes nécessaires). Il est fort probable qu'une nouvelle version officielle sera distribuée avant longtemps. Pour le compilateur grenoblois nous nous orientons vers la version approuvée à la dernière réunion du WG 2.1 [24] avec l'espoir que celle-ci ne sera pas trop éloignée de la version finale.

Ces changements sont sensibles au niveau du problème de détermination des modifications (en effet le problème est devenu un peu plus simple). Mais pour cette thèse la version officielle sera utilisée. Il y a

essentiellement deux raisons à cela :

(1) - le rapport ALGOL 68 original a été publié et est donc stable en tant que document de référence.

(2) - c'est sur ce langage-là que le travail a été initialement réalisé.

5.1 - Contraintes de contexte en ALGOL 68 -

On a très rigoureusement défini la notion de conversion de type de données en ALGOL 68. Dans le rapport elles sont appelées modifications. Il y en a huit dans la version officielle :

(1) - déprocédurer : la valeur (de mode proc ...) est appelée comme une procédure. Le mode de la valeur retournée est le mode résultant de la modification (c'est-à-dire le mode résultat de la procédure). (ex : proc ent $\xrightarrow{\text{déproc}}$ ent)

(2) - dérepérer : la valeur (de mode rep ...) est l'adresse (ALGOL 68 : nom) d'une autre valeur. Celle-ci est le résultat de l'application de la modification.(ex : rep bool $\xrightarrow{\text{dérep}}$ bool)

(3) - procédurer : l'application de la modification à une valeur de mode m, produit une routine dont le résultat est une valeur de mode m. Le mode résultant de la modification est proc m.
(ex : réel $\xrightarrow{\text{procer}}$ proc réel)

(4) - unir : l'application de la modification a une valeur de mode m, produit une valeur de mode union (..., m, ...) contenant le premier mode comme un de ses composants. C'est-à-dire que m est un des modes possibles pour cette union.
(ex : bool $\xrightarrow{\text{unir}}$ union (réel, ent, bool))

(5) - élargir : la valeur d'un certain mode est modifiée dans la valeur mathématiquement équivalente d'un autre mode. La modification s'applique seulement quand une correspondance exacte existe.
(ex : ent $\xrightarrow{\text{élargir}}$ compl mais pas compl à ent)

(6) - ranger : la valeur a un certain mode, m. La modification crée une valeur qui est un tableau (ALGOL 68 : rang) contenant la valeur originale comme élément unique. De la même façon, une adresse d'une valeur peut devenir l'adresse d'un tableau de la même valeur.
(ex : car $\xrightarrow{\text{ranger}}$ [ ] car ou rep car $\xrightarrow{\text{ranger}}$ rep [ ] car)

(7) - hisser : l'opérande de la modification est une construction syntaxique de mode neutre. Il y a trois possibilités : le allera, le fant ou le nil. La modification crée un objet du mode désiré (quel qu'il soit), ayant une valeur indéfinie.
(ex : neutre $\xrightarrow{\text{hisser}}$ rep proc ent)

(8) - neutraliser : la valeur a n'importe quel mode m. Le résultat de la modification est toujours du mode neutre. Cela revient à abandonner la valeur originale.
(ex : compl $\xrightarrow{\text{neutr'er}}$ neutre)

Un mélange de ces opérations peut être appliqué, l'une après l'autre, à la même construction syntaxique. Si le langage ALGOL 68 est un langage bien défini, il est clair qu'il ne doit y avoir qu'une seule séquence de modifications autorisée, pour toute position syntaxique donnée. En conséquence des contraintes doivent être introduites afin d'interdire les applications abusives de modifications comme par exemple :

de procédurer après une modification procédurer
hisser après une modification neutraliser ... etc....

Les contraintes nécessaires dépendent de la nature de la position syntaxique en question. Ces positions se divisent en quatre classes, chacune ayant comme principe, de permettre le nombre maximum de modifications possibles sans créer d'ambiguïtés. Ces quatre classes (que j'appelle "puissances") ont, dans l'ordre croissant des contraintes, les forces de contextes : "forte", "forme", "faible" et "molle".

5.11 - Les contraintes de la position molle -

Il n'y a qu'une seule position molle dans les constructions syntaxiques d'ALGOL 68 : la partie gauche d'une affectation. Le mode final de la partie gauche doit toujours commencer avec rep, car sa valeur est l'adresse où doit être stockée la valeur de la partie droite. Par exemple, si la valeur à stocker est du mode réel, l'identificateur ou expression à gauche doit donner une valeur rep réel indiquant où doit être rangée la valeur réelle.

Parce qu'il s'agit d'une affectation, on peut déduire le genre des modifications permises dans une position molle. En effet, il n'y en a qu'une seule : le déprocédurage. On peut, par exemple, avoir en partie gauche un identicateur de mode proc rep compl. La routine représentée par l'identificateur est appelée en faisant le déprocédurage. Le résultat retourné par cette routine est une valeur de mode rep compl, c'est-à-dire une adresse que l'on peut utiliser pour stocker une valeur complexe. Maintenant regardons pourquoi les autres modifications doivent être interdites.

Les modifications procédurer, élargir, ranger et neutralisation peuvent être éliminées tout de suite pour la raison très simple qu'aucune ne peut conduire vers un mode commençant par rep. Ces modifications sont donc inutiles.

La modification hisser pourrait certainement produire un mode commençant avec rep, en fait elle peut en produire un nombre quelconque : rep ent, rep rep bool, etc.... Donc la modification hisser doit être interdite à cause des ambiguïtés qu'elle peut produire.

Il ne reste que la modification dérepérer. On voit vite que l'on peut produire un rep ... mode avec cette modification. Par exemple un rep rep ent peut être dérepéré vers un rep ent. (La valeur de mode rep ent est simplement l'adresse d'un entier). Ici encore il y a l'ambiguïté. Si le mode original à gauche d'une affectation est un rep rep ent, le mode final peut être aussi bien rep ent que rep rep ent si la modification dérepérer est permise. Donc elle est interdite.

5.12 - Les contraintes de la position faible -

La position faible correspond, elle aussi, à très peu de constructions ALGOL 68. Il y en a deux :

(a) - un indexage d'un tableau.
(ex : a[i, j])

(b) - une sélection d'une structure.
(ex : re de Z)

(Les parties indiquées par ⌊⌋ sont en position faible). Donc on peut déduire que le mode final doit être ou bien [ ] ... ou bien struct (...). En fait il existe encore deux possibilités. Il se peut que ce soit l'adresse du tableau ou de la structure qui soit désirée. Dans ce cas, le mode final peut être aussi, rep [ ]... ou rep struct (...).

Avec ces possibilités, considérons quelles sont les modifications à éliminer. Essentiellement toutes sont exclues, sauf déprocédurer et dérepérer, pour les mêmes raisons que dans le cas de la position molle.

Déprocédurer est permis comme auparavant mais dérepérer pose certains problèmes. Parce que les modes struct (...) et rep struct (...) peuvent tous les deux être des modes finaux, il y a un danger d'ambiguïté évident en permettant le déreperage de rep struct (...). Pour éviter ce problème et l'analogue pour les tableaux, on interdit le dérepérage du dernier rep avant la structure ou le tableau. Notons que le dérepérage de rep rep [ ] m à rep [ ] m ne posent aucun problème, rep [ ] m étant maintenant le seul mode final possible dans ce contexte. Cette limitation de la modification dérepérer en position faible est appelée "modification dérepérer en position faible".

En résumé, les constructions en position faible permettent seulement de déprocédurer et de dérepérer.

5.13 - Les contraintes de la position ferme -

La seule construction significative en position ferme est celle d'opérande d'une expression (ALGOL 68 : formule). Par exemple, les positions soulignées dans l'exemple suivant sont en contexte ferme :

a + b * e

Chaque opérateur monadique ou dyadique d'un programme ALGOL 68 attend pour ses opérandes certains modes bien précis. Par exemple, l'opérateur plus (+) peut attendre des entiers comme opérandes. Si l'expression qui est un de ces opérandes délivre une valeur avec mode rep rep ent, deux modifications dérepérer doivent être appliquées pour arriver à l'entier désiré.

Ce qui complique le problème est qu'un opérateur donné peut attendre pour chacun de ses opérandes, n'importe lequel de toute une liste de modes. C'est cette possibilité qui implique la plupart des contraintes imposées sur la position ferme.

En particulier, la modification élargir ne peut être permise à cause des ambiguïtés qui seraient provoquées. Supposons que l'opérateur "plus" attend réel ou ent comme mode d'un opérande. L'opérande effectif ayant une valeur originale de mode ent. Si on permet d'élargir cet ent en réel, on ne sait pas si l'opérateur plus désiré est celui entre entiers ou celui entre réels. Donc la modification élargir en position ferme doit être interdite.

Pour exactement la même raison, la modification ranger est interdite. Un opérateur peut attendre comme opérande un mode m ou un mode [ ] m. Si le mode m d'un opérande peut être modifié en un mode [ ] m, l'instance de l'opérateur à choisir ne peut être déterminée.

La modification hisser est hors de question car un mode original peut être hissé en un quelconque mode appartenant à la liste de modes des opérandes. Donc il serait impossible de choisir quel est l'opérateur à utiliser.

La modification neutraliser est éliminée pour la raison très simple qu'elle est inutile. En effet, le mode neutre ne peut pas être le mode attendu d'un opérateur.

On pourrait penser, à ce point, que toutes les modifications sont interdites pour cause d'ambiguïtés dans le choix de l'opérateur. En réalité, toutes les autres modifications (déprocédurer, dérepérer, procédurer, unir) sont permises. Cela est uniquement du au fait que les listes des modes des opérandes sont elles-aussi contraintes de manière à permettre les autres modifications. Par exemple, un opérateur monadique ne pourrait pas attendre réel et rep réel à la même fois, car sinon la possibilité de dérepérer permettrait un choix ambigu d'opérateur.

Le choix des modifications permises en contexte ferme a certainement un côté arbitraire. Il n'y a rien dans les constructions syntaxiques qui oblige ce choix plutôt qu'un autre. Comme il est normal dans ces situations là, les modifications admises sont celles conduisant dans les cas classiques, à l'interprétation la plus "naturelle". (Jugement des auteurs du Rapport).

5.14 - Les contraintes de la position forte -

Chacune des forces de contexte déjà discutées (molle, faible, et fermé) étaient limitées à très peu de constructions ALGOL 68. La position forte couvre tout le reste. Par exemple, la partie droite d'une affectation d'un appel, la partie booléenne d'une proposition conditionnelle, etc....

Le mode attendu dans une position forte est toujours complètement connu. Contrairement au cas de la position ferme, il n'y a toujours qu'un seul mode attendu, pour une construction donnée. Dans certains cas le mode attendu peut être neutre. Le mode original et le mode final (attendu) étant tous les deux totalement spécifiés, toutes les modifications sont permises dans cette position. Bien entendu, certaines combinaisons de modifications doivent toujours être exclues pour éviter les ambiguïtés. Ces problèmes seront discutés dans la Section 5.224 sur la détermination des modifications en position forte.

5.2 - Détermination des modifications documentées par macros-syntaxiques -

La description du processus de détermination des modifications qui va suivre est le développement d'une analyse soignée des règles syntaxiques du Rapport. L'objectif poursuivi, ici, est de présenter l'algorithme de détermination sous une forme suggèrant une implantation. Par souci de précision, la description de chaque pas sera écrite en macros-syntaxiques.

5.21 - La méthode -

Le problème se divise nettement en deux parties :

(1) - La détermination de la liste des modifications à appliquer à une construction particulière, étant donné la force du contexte, le mode original (à priori) et au moins un préfixe du mode final (à postériori). Ce processus sera appelé l'"algorithme de liste des modifications".

(2) - La détermination des modes et force contextuelle pour chaque construction syntaxique d'un programme, afin de permettre l'application de l'algorithme de liste des modifications. Les forces de contexte et les modes pouvant être transmis d'une construction à une autre, il est nécessaire d'avoir un algorithme capable de traverser le texte intermédiaire en appliquant l'algorithme de liste des modifications, chaque fois que tous les paramètres requis ont été rassemblés.

La première partie du problème sera exposée dans Section 5.22 et la deuxième dans 5.23. Ensuite dans 5.24 le problème, lié au précédent, du choix de l'exemplaire d'un opérateur sera examiné (ex : le choix entre multiplication entre entiers ou entre réels). Comme conclusion, une évaluation de l'utilisation des macros-syntaxiques, pour la définition de tels algorithmes, sera faite en Section 5.25.

5.22 - L'algorithme de liste des modifications -

Dans le compilateur grenoblois les segments de texte sont traduits en arbres d'opérateurs juste avant d'être utilisés pour la détermination des modifications et la production de code. Dans les algorithmes qui vont suivre, les arbres d'opérateurs seront représentés par une grammaire polonaise préfixée :

```
noeud → feuille |
        opérateur•unaire noeud |
        opérateur•binaire noeud noeud | ...
```

(N.B. : La manière de traduire ALGOL 68 infixé en ALGOL 68 préfixé est essentiellement la même que celle décrite pour PL/1).

Le mot "opérateur" va être utilisé avec deux sens différents. Là où le sens ne sera pas clair, j'écrirai "opérateur de l'arbre" pour l'opérateur d'un noeud de l'arbre et j'écrirai "opérateur ALGOL 68" pour les opérateurs définis dans le Rapport ALGOL 68, c'est-à-dire, les symboles +, *, etc... et les opérateurs définis par les déclarations de l'utilisateur.

Dans la présentation de l'algorithme de liste des modifications, les modifications seront traitées comme si elles étaient de nouveaux opérateurs ALGOL 68. Leur représentation, dans l'arbre des opérateurs, conduit vers l'expansion suivante du non-terminal <opérateur unaire> de la grammaire préfixée :

```
opérateur•unaire → 'dérep' | 'déproc' | 'proced' |
                   'unir' | 'élargir' | 'neutr'er' |
                   'ranger' | 'hisser'
```

Par exemple, regardons l'instruction ALGOL 68 suivante :

```
... ; x := y := z ; ...
```

x, y et z sont tous du mode rep réel.

On doit dérepérer z avant que la valeur résultante puisse être affectée à l'adresse indiquée par y. De la même façon y est dérepéré pour obtenir cette même valeur, à stocker dans l'adresse donnée par x. La valeur de toute l'instruction qui est la même que la valeur de z (rep réel) est abandonnée au point virgule. Ceci est indiqué, par la modification neutraliser. Donc le problème essentiel de l'algorithme est de lire le texte original ci-dessus et de l'analyser comme si on avait écrit :

... ; neutrer (x := dérep (y := dérep z)) ; ...

Avec une seule exception (neutraliser), les règles pour l'établissement de la liste des modifications ne dépendant pas du type de noeud-opérateur mais simplement de la force du contexte. Ces règles changent beaucoup, d'une force de contexte à une autre, chaque cas sera traité individuellement ci-dessous.

5.221 - La liste des modifications en position molle -

Le mode final, dans un contexte mou, doit toujours commencer par un rep (car, il s'agit de la partie gauche d'une affectation). La modification déprocédurer étant la seule, le mode original dans cette position doit toujours être de la forme :

$$\text{proc}_1 \cdots \text{proc}_n \text{ rep} \ldots \qquad (n \geq 0)$$

Par exemple, si le mode original est proc rep [ ] réel, le mode final est forcement rep [ ] réel après une seule modification déprocédurer. Dans le programme, celà signifie que la routine associée à la partie gauche de l'affectation est appelée et qu'elle retourne une valeur de mode rep [ ] réel qui fournit l'adresse où stocker le [ ] réel délivré par la partie droite.

Pour réaliser cette procédure avec des macros-syntaxiques, supposons que l'on a ajouté l'indicateur spécial "mou" et le mode original (à priori), dans le texte, juste avant le noeud-opérateur à traiter. Ces deux éléments correspondent aux paramètres effectifs de l'algorithme de liste des modifications.

La macro est :

noeud → 'mou' mode$_{pre}$ noeud$_s$

= si mode$_{pre}$ → 'proc' mode$_{pre2}$

alors 'mou' mode$_{pre2}$ 'déproc' noeud$_s$

sinon 'terminé' mode$_{pre}$ noeud$_s$

fsi

Cette macro traduit, par exemple, le texte :

mou (proc proc rep réel)$_{mode_{pre}}$ (a)$_{noeud_s}$

en :

mou (proc rep réel)$_{mode_{pre}}$ (déproc a)$_{noeud_s}$

puis en :

mou (rep réel)$_{mode_{pre}}$ (déproc déproc a)$_{noeud_s}$

et enfin en :

terminé rep réel déproc déproc a

L'indicateur "terminé" sert à conserver le mode final pour vérification et utilisation ailleurs dans l'arbre des opérateurs (ceci est le deuxième algorithme, décrit en Section 5.23).

Il est utile de remarquer que, lors de chaque appel récursif de la macro ci-dessus, la définition de <noeud>$_s$ change. Les opérateurs de modification lui seront ajoutés à chaque pas.

5.222 - La liste des modifications en position faible -

Dans un contexte faible, le mode final (à postériori) peut avoir une des quatre formes

struct (...) [...] ...

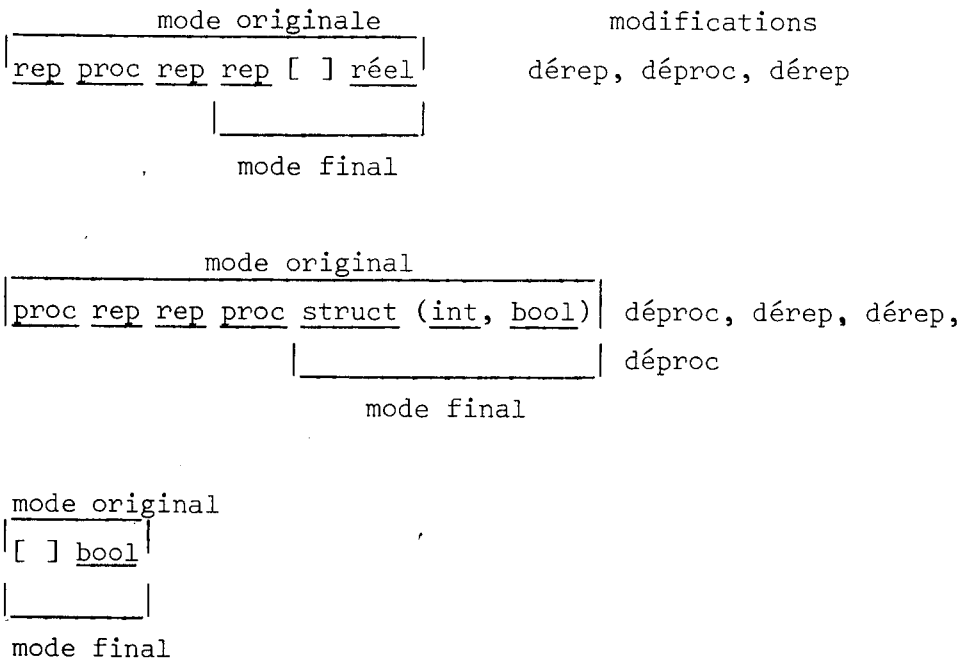
rep struct (...) rep [...] ...

Les modifications déprocédurer et "dérepérer en position faible" (le dernier rep avant la structure ou le tableau ne doit pas être supprimé) sont les seules permises. Ceci implique que le mode original a la forme :

$$\{\underline{\text{rep}} \mid \underline{\text{proc}}\}^n \{\underline{\text{struct}}\ (\ldots) \mid [\ ]\ \ldots\} \quad (n \geq 0)$$

L'algorithme pour déterminer la listes des modifications et le mode final prend le mode original et enlève successivement les symboles rep et proc de tête jusqu'à obtenir un mode final acceptable.

Exemples :



La macro a essentiellement la même apparence que celle du contexte mou, excepté qu'il faut considérer les deux modifications dérepérer et déprocédurer et qu'un test spécial est nécessaire pour vérifier que la modification dérepérer est permise.

noeud → 'faible' $mode_{pre}$ $noeud_{simple}$

= si $mode_{pre}$ → 'proc' $mode_2$

alors 'faible' $mode_2$ 'déproc' $noeud_{simple}$

sinsi $mode_{pre}$ → 'rep' $mode_2$

alors si ($mode_2$ → 'rep' $mode_3$) v ($mode_2$ → 'proc' $mode_3$)

alors 'faible' $mode_2$ 'dérep' $noeud_{simple}$

sinon 'terminé' $mode_{pre}$ $noeud_{simple}$

fsi

sinon 'terminé' $mode_{pre}$ $noeud_{simple}$

fsi

Un exemple d'application serait :

faible |rep proc rep [ ] réel| |a|
$mode_{pre}$ $noeud_{simple}$

=> faible |proc rep [ ] réel| |dérep a|

=> faible |rep [ ] réel| |déproc dérep a|

=> terminé |rep [ ] réel| |déproc dérep a|
mode à postériori noeud final

Remarque :

La dernière modification dérepérer n'a pas été réalisée car le test :

($mode_2$ → 'rep' $mode_3$) v ($mode_2$ → 'proc' $mode_3$)

n'a pas marché. $<mode>_2$ était en fait

$$mode_2 \rightarrow \text{'[ ]'}\ mode_3$$

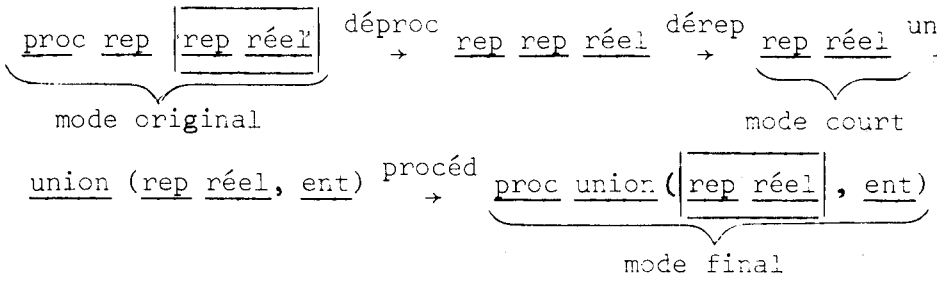
Ceci correspond avec la définition de la modification dérepérer en position faible.

5.223 - La liste des modifications en position ferme -

En contexte ferme, le mode final est toujours connu car il appartient à la liste des modes permis pour l'opérateur ALGOL 68 concerné. En plus des modifications déprocédurer et dérepérer, les modifications procédurer et unir sont aussi permises.

Quoique cela ne soit pas apparent à la lecture de leur définition, ALGOL 68 sépare ces deux groupes de modifications. En effet, toutes les modifications dérepérer et déprocédurer doivent être appliquées sur un noeud avant les modifications unir et procédurer. Une fois appliquée une de ces dernières, elles sont alors les seules applicables pour obtenir le mode désiré.

Cette séparation a une conséquence très utile pour l'implémentation. Remarquons que les deux modifications dérepérer et déprocédurer raccourcissent le mode original par suppression des symboles rep et proc respectivement. Inversement les modifications procédurer et unir rallongent le mode par addition des symboles proc et union (...) jusqu'à l'obtention du mode final. En conséquence, il existe au moins un mode qui est composant, à la fois, du mode original et du mode final. Si l'on prend le plus long de ces modes, on a alors un mode intermédiaire dans le passage du mode original au mode final. Du fait que ce mode est le plus court des modes apparaissant dans ce cheminement, nous l'appelerons le "mode court". Considérons l'exemple suivant du passage d'un mode original à un mode final.



Une fois le mode court connu, les modifications à appliquer peuvent être déterminées en regardant simplement les préfixes du mode original et du mode final.

Tout d'abord le préfixe du mode original est analysé de gauche à droite et les modifications déprocédurer et dérepérer sont déterminées. Puis le préfixe du mode final est analysé de droite à gauche et les modifications procédurer et unir sont déterminées. Il faut faire attention à un détail lors de la détermination du mode court. Comme indiqué, dans sa définition, la modification unir peut-être utilisée simplement pour étendre une union en une autre union contenant les composants de la première. C'est-à-dire :

$$\underline{\text{union}}(\underline{\text{m1}}, \ldots, \underline{\text{m}_n}) \overset{\text{unir}}{\rightarrow} \underline{\text{union}}(\text{m1}, \ldots, \underline{\text{m}_n}, \ldots)$$

Pour le mode court, on doit considérer la première union comme contenue dans la seconde.

Exemple :

$$\underbrace{\underline{\text{rep}}\ \underline{\text{union}}\ (\underline{\text{réel}}, \underline{\text{ent}})}_{\text{mode original}} \overset{\text{dérep}}{\rightarrow} \underbrace{\underline{\text{union}}\ (\underline{\text{réel}}, \underline{\text{ent}})}_{\text{mode court}} \overset{\text{unir}}{\rightarrow} \underbrace{\underline{\text{union}}\ (\underline{\text{réel}}, \underline{\text{ent}}, \underline{\text{bool}})}_{\text{mode final}}$$

Comme on peut s'en douter, à partir de la discussion précédente, la macro-syntaxique pour le contexte ferme est sensiblement différente des précédentes.

Premièrement, l'algorithme des modifications demande que les modes à priori et à postériori soient connus. Le texte initial doit donc être :

$$\underline{\text{ferme}}\ \text{mode}_{\text{post}}\ \text{mode}_{\text{pre}}\ \text{noeud}_s$$

Deuxièmement, il n'y a pas besoin de retourner le mode à postériori puisqu'il est déjà connu. Donc, le mot-clé "terminé" n'est pas nécessaire. Quand la détection des modifications est terminée (c'est-à-dire quand le postmode est devenu identique au prémode) le mot-clé "ferme" est simplement éliminé.

La macro sera plus facile à comprendre si on étudie, d'abord, un exemple des transformations désirées.

Considérons l'exemple du début de ce paragraphe.

(contexte et opérateurs)	(mode à postériori)	(mode à priori)	(opérateurs)	(noeud)
=> ferme -	proc union (rep réel, ent)	proc rep rep réel	-	idf
=> ferme -	proc union (rep réel, ent)	rep rep réel	déproc	idf
=> ferme -	proc union (rep réel, ent)	rep réel	dérep déproc	idf
=> procéd ferme	union (rep réel, ent)	rep réel	dérep déproc	idf
=> procéd unir ferme	rep réel	rep réel	dérep déproc	idf

=> procéd unir dérep déproc idf

A chaque fois qu'une modification est détectée elle est placée soit à droite, soit à gauche de la transformation, suivant que le mode réduit est le mode à priori ou le mode à postériori.

Toutes les transformations possibles sont schématisées dans le tableau ci-dessous :

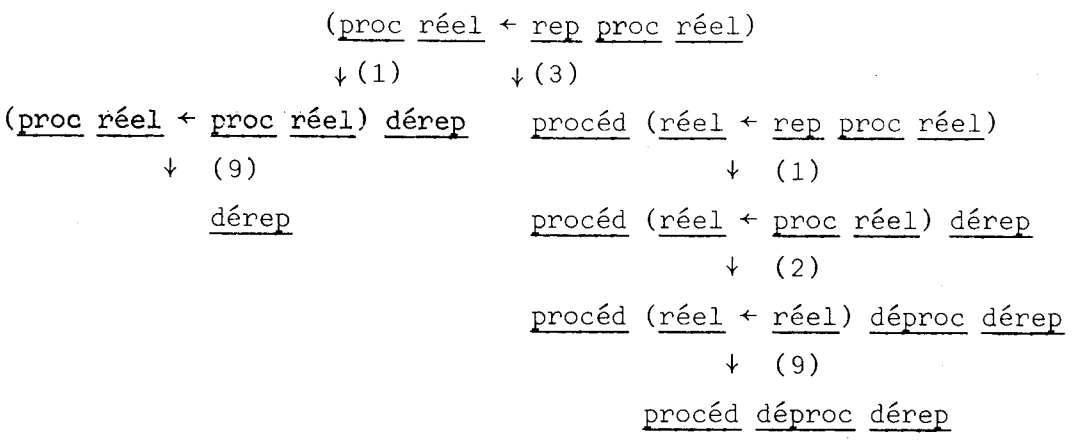
	postmode	prémode		modification allongeante	modes restants		modification raccourcissante
(1)	$mode_{post}$	rep $mode_{pre2}$	⇒	-	$mode_{post}$	$mode_{pre2}$	dérep
(2)	$mode_{post}$	proc $mode_{pre2}$	⇒	-	$mode_{post}$	$mode_{pre2}$	déproc
(3)	proc $mode_{post2}$	$mode_{pre}$	⇒	proced	$mode_{post2}$	$mode_{pre}$	-
(4)	union (m, mlist)	$mode_{pre}$	⇒	unir	m	$mode_{pre}$	-
(5)				-	union (mlist)	$mode_{pre}$	-
(6)	union (m_2, m_3)	$mode_{pre}$	⇒	unir	m_2	$mode_{pre}$	-
(7)				unir	m_3	$mode_{pre}$	-
(8)	union ($mlist_{post}$)	union ($mlist_{pre}$)	⇒	unir	-	-	- (*)
(9)	$mode_{post}$	$mode_{pre}$	⇒	-	-	-	- (**)

(*) si et seulement si le prémode est une sous-union du postmode

(**) si et seulement si les deux modes sont identiques.

Les transformations (8) et (9) terminent toute série réussie de transformations en éliminant les modes.
Les autres transformations s'appliquent quand les conditions indiquées par les deux premières colonnes sont remplies. Remarquons qu'il peut y avoir plusieurs transformations applicables dans un même état. Soit le résultat final sera identique et, en conséquence, ignoré (cf. "ambiguïté en position faible", § 1.8), soit les séries issues d'une application incorrecte se bloqueront un peu plus loin, par impossibilité d'appliquer d'autres transformations.

Il y a une exception à l'affirmation qu'il n'existe qu'un seul résultat correct. Soit l'exemple :



La série de transformations de gauche est la bonne. Celle de droite, quoique correcte d'après la table de transformations précédente, est interdite par la syntaxe d'ALGOL 68 (on ne peut pas appliquer à une valeur la modification procédurer juste après la modification déprocédurer). Ce cas peut être facilement éliminé en vérifiant que la séquence procéd déproc n'apparaît jamais dans une liste de modifications.

La macro définissant la détection des modifications peut maintenant être écrite. La métanotation

remplacement 'ou' remplacement

signifie que les deux remplacements sont à essayer en parallèle.

La macro ci-dessous n'est rien d'autre qu'une réécriture formalisée de la table de transformations.

```
noeud → 'ferme' mode_post mode_pre noeud_s

    = si mode_post = mode_pre

      alors noeud_s

      sinsi est-sousunion (mode_post, mode_pre)

      alors 'unir' noeud_s

      sinon si mode → 'rep' mode_pre2

            alors 'ferme' mode_post mode_pre2 'derep' noeud_s

            sinsi mode_pre → 'proc' mode_pre2

            alors 'ferme' mode_post mode_pre2 'déproc' noeud_s

            fsi
         ou
            si mode_post → 'proc' mode_post2

            alors 'procéd' 'ferme' mode_post2 mode_pre noeud_s

            sinsi mode_post → 'union(' modelist ')'

            alors si modelist → mode_2 ',' modelist_2

                  alors 'unir' 'ferme' mode_2 mode_pre noeud_s
                     ou
                        'ferme' 'union(' modelist_2 ')' mode_pre noeud_s

                  sinsi modelist → mode_2 ',' mode_3

                  alors 'unir' 'ferme' mode_2 mode_pre noeud_s
                     ou
                        'unir' 'ferme' mode_3 mode_pre noeud_s

                  fsi

         fsi

      fsi
```

On peut se demander comment est faite la vérification qu'il n'y a pas de modification procédurer avec déprocédurer. La seule position, ou elle doit apparaître, est lors de l'application de la transformation (9). Le préfixe du prémode produit des modifications déprocédurer et le préfixe du postmode, des modifications procédurer.
Le seul contact entre ces deux chaînes d'opérateurs intervient quand les modes disparaissent. Donc, en principe, quand le prédicat

$$mode_{post} = mode_{pre}$$

est vrai, il suffit d'examiner les deux opérateurs voisins (l'opérateur final de la première liste et l'opérateur initial de la deuxième). Cependant, l'opérateur gauche se trouve en dehors de l'appel de la macro

'ferme' $mode_{post}$ $mode_{pre}$ $noeud_s$

Pour faire ce test, il serait nécessaire d'introduire une fonction prédicat pour accéder à l'opérateur gauche.

Cette fonction n'a pas été introduite dans notre implémentation présente.

Il est, cependant, assez facile de faire ce test indirectement. Remplaçons la tête de la macro par :

noeud → 'ferme' $mode_{post}$ $mode_{pre}$ $noeud_s$

= si $mode_{post} = mode_{pre}$

alors 'test' $noeud_s$

sinsi ...

Le mot-clé 'test' peut être détecté par une autre macro qui inclut spécifiquement les contextes droit et gauche.

noeud → op-unaire$_g$ 'test' op-unaire$_d$

= si (op-unaire$_g$ → 'procéd') ∧ (op-unaire$_d$ → 'déproc')

alors échec

sinon op-unaire$_g$ op-unaire$_d$

fsi

Cette macro bloque la traduction, à moins que la condition pour supprimer 'test' soit remplie.

5.224 - La liste des modifications en position forte -

Comme le montre la discussion précédente, la détection des modifications pour le contexte fort est pratiquement une extension de celle pour le contexte ferme. En effet, on peut, simplement, étendre la table de transformation précédente, la transcription en macros étant, alors, quasi automatique.

5.2241 - MODIFICATION RANGER -

Deux transformations sont nécessaires, une pour le cas rang, l'autre pour le cas rep rang.

(10) ⌊rang mode$_{post2}$⌋ ⌊mode$_{pré}$⌋ => ranger ⌊mode$_{post2}$⌋ ⌊mode$_{pré}$⌋ -

(11) ⌊rep rang mode$_{post2}$⌋ ⌊rep mode$_{pré2}$⌋ => ranger ⌊rep mode$_{post2}$⌋⌊rep mode$_{pré2}$⌋ -

Un exemple serait :

fort (rep rang rang proc réel ← rep rep proc réel) idf
↓
fort (rep rang rang proc réel ← rep proc réel) dérep idf
↓
ranger fort (rep rang proc réel ← rep proc réel) dérep idf
↓
ranger ranger fort (rep proc réel ← rep proc réel) dérep idf
↓
ranger ranger dérep idf

5.2242 - MODIFICATION ELARGIR -

Une transformation est nécessaire pour chaque type de modifications élargir possible. Remarquons qu'il s'agit toujours de transformations terminales.

(12) $\{long\}_o^n$ réel $\{long\}_o^n$ ent => élargir

(13) compl ent => élargir élargir

(14) compl réel => élargir

(15) rang bool $\{long\}_o^n$ bits => élargir

(16) rang char $\{long\}_o^n$ bytes => élargir

Soit l'exemple :

fort (proc compl ← rep rep ent) idf
↓ (1)
fort (proc compl ← rep ent) dérep idf
↓ (1)
fort (proc compl ← ent) dérep dérep idf
↓ (3)
procéd fort (compl ← ent) dérep dérep idf
↓ (13)
procéd élargir élargir dérep dérep idf

5.2243 - MODIFICATION NEUTRALISER -

Comme la nature de la modification neutraliser diffère suivant que le noeud original est une confrontation ou non, une transformation préliminaire doit être appliquée.

(17) neutre $mode_{pré}$ $\Rightarrow$ { confneutre $mode_{pré}$ (si et seulement si confrontation) ; autreneutre $mode_{pré}$ (si et seulement si non confrontation) }

La modification neutraliser pour une confrontation est simplement :

(18) confneutre $mode_{pré}$ $\Rightarrow$ neutraliser

Remarquons que ces transformations doivent être appliquées de préférence aux autres et non en parallèle comme dans les autres cas. Ceci s'applique aussi au reste des transformations concernant la modification neutraliser

Voici un exemple d'une confrontation neutralisée :

fort (neutre ← rep proc ent) ⌊rpe := pe⌋
↓ (17)
fort (confneutre ← rep proc ent) ⌊rpe := pe⌋
↓ (18)
neutraliser ⌊rpe := pe⌋

Les autres modifications neutraliser posent un problème particulier, car les symboles proc et rep doivent être enlevés du mode à priori jusqu'au dernier symbole proc avant de pouvoir appliquer la modification neutraliser. Par exemple :

Les opérateurs, résultant des modifications du mode à priori

rep proc rep proc ent,

sont

neutraliser déproc dérep déproc dérep

et non

neutraliser déproc dérep

Pour forcer celà, le symbole vtest est ajouté quand une modification déprocédurer est appliquée. Ce symbole ne sera éliminé que si aucun autre symbole proc n'est rencontré durant les transformations suivantes.

(19)	autreneutre	proc $mode_{pré2}$	⇒	autreneutre	$mode_{pré2}$	déproc
(20)				vtest	$mode_{pré2}$	déproc
(21)	autreneutre	rep $mode_{pré2}$	⇒	autreneutre	$mode_{pré2}$	dérep
(22)	vtest	rep $mode_{pré2}$	⇒	vtest	$mode_{pré2}$	-
(23)	vtest	$mode_{pré}$	⇒	neutraliser	(si et seulement si $mode_{pré} \neq$ proc ... $\wedge$ $mode_{pré} \neq$ rep ...)	

Considérons d'abord un exemple simple :

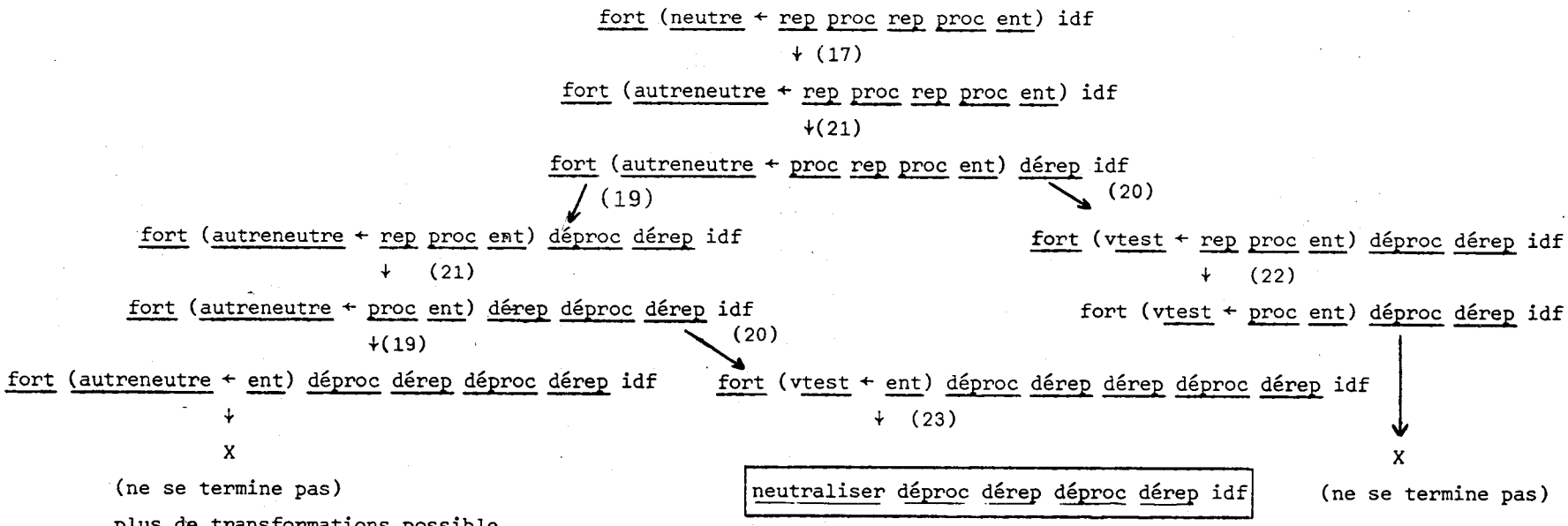
fort (neutre ← rep proc ent) rpe
↓ (17)
fort (autreneutre ← rep proc ent) rpe
↓ (21)
fort (autreneutre ← proc ent) dérep rpe
↙ (19) ↘ (20)

fort (autreneutre ← ent) déproc dérep rpe
↓
X
(ne se termine pas)

fort (vtest ← ent) déproc dérep rpe
↓ (23)
neutraliser déproc dérep rpe

Il est intéressant de comparer cet exemple avec celui pour la confrontation avec le même mode à priori. Dans le premier cas la valeur (de mode proc ent) était simplement affectée et le résultat (de mode rep proc ent) implicitement abandonné grâce à la coercion neutraliser. Par contre, dans le cas d'un identificateur simple, le résultat n'est pas abandonné mais dérepéré et ensuite déprocéduré. La valeur entière obtenue est alors abandonnée.

Voici maintenant un exemple qui exploite plus clairement le symbole vtest.



Le symbole vtest bloque chaque série qui ne peut pas produire une opération neutraliser après la dernière modification déprocédurer possible.

5.2244 - MODIFICATION UNIR -

Le changement nécessaire pour empêcher l'application des modifications fortes à l'intérieur d'une union, même si le contexte original est fort, est actuellement très simple. Il suffit de changer le mot-clé fort par le mot-clé ferme quand il est directement associé à un mode à postériori de type union. Cela entraîne l'utilisation de la macro des modifications fermes, pour le reste des transformations.

(24) fort union (mlist) $mode_{pré}$ => ferme union (mlist) $mode_{pré}$

$mode_{post}$ $mode_{post}$

5.225 - Résumé -

On vient de montrer comment les listes de modifications peuvent être déterminées par des séries appropriées de transformations par macros syntaxiques. Dans le cas mou et faible, les transformations commencent avec le mot-clé approprié et le mode à priori et se terminent avec "terminé", le mode à postériori et la liste des modifications, exprimées comme des opérateurs ALGOL 68. Dans les cas ferme et fort, les transformations commencent avec le mot-clé et les modes à priori et à postériori et se terminent avec la liste des modifications.

Il faut maintenant discuter le mécanisme pour déterminer, à chaque noeud, sa force de contexte et ses modes. C'est le sujet du reste de ce chapitre.

5.23 - Contextes et modes pour les modifications -

Une fois connue la routine de base servant à déterminer la liste des modifications à appliquer à un noeud opérateur particulier, le seul problème restant est de savoir comment trouver les paramètres pour appliquer cette routine. En utilisant le modèle d'arbre d'opérateurs, qui correspond à la fois à l'implémentation en cours à Grenoble et à la technique de définition des macros-syntaxiques, le processus commence au noeud

racine avec le contexte "fort" et le mode à postériori "neutre".

Cette information est propagée au travers de l'arbre de manière à assurer que tous les noeuds aient leurs modifications déterminées.

En regard du problème de détection des modifications, les noeuds de l'arbre ont certaines caractéristiques générales.

Ces noeuds sont :

a) - *créateurs de contexte* - c'est-à-dire qu'ils établissent le contexte pour un ou plusieurs de leurs sous-noeuds (ex : le noeud racine, le noeud affectation)

b) - *déterminants* - c'est-à-dire qu'ils permettent d'établir leur liste de modification sans l'aide de leurs sous-noeuds (ex : le noeud identificateur)

c) - *transmetteurs* - c'est-à-dire qu'ils transmettent un contexte à un ou plusieurs de leurs sous-noeuds qui, plus tard, leur retourneront des résultats (ex : le noeud point-virgule)

Ces caractéristiques ne s'excluent pas mutuellement. Par exemple, le noeud "si-alors-sinon" est, à la fois, créateur de contexte (pour la clause booléenne en contexte fort) et transmetteur (car le contexte soit de la partie "alors", soit de la partie "sinon" dépend du contexte avec lequel la clause conditionnelle a été trouvée).
Le noeud "est" est à la fois déterminant (il doit avoir un mode à postériori booléen, quels que soient ses sous-noeuds) et créateur de contexte (un sous-noeud a un contexte mou et l'autre a un contexte fort).

Un processus typique de détection des modifications peut être décrit comme suit :

Partant d'un noeud déterminant, un contexte et un mode à postériori sont spécifiés pour le sous-noeud à traiter. Si le contexte est faible

ou mou, alors le mode à postériori n'est pas encore connu et il doit donc être remplacé par le symbole inconnu. En termes de macro-syntaxiques cela signifie que la macro, définissant les opérations pour le noeud, insérera une phrase du type :

(∇ fort bool)

en-tête du sous-noeud à traiter. Le caractère spécial ∇ déclenchera l'appel de la macro traitant le sous-noeud. Toutes ces macros sont appelées avec la même forme générale :

noeud → '∇' contexte $mode_{post}$ 'opr_x' $noeud_1$ $noeud_2$ ...

"opr_x" est le nom de l'opérateur.

Si le sous-noeud à traiter est un noeud transmetteur, le contexte et le mode à postériori est alors simplement transmis en dessous. Il se peut que le mode soit modifié avant d'être passé. Dans les macros-syntaxiques, la phrase de remplacement sera formée du marqueur ∇ suivi du contexte et du mode.

En fin de compte, les "transmissions" s'arrêtent à un noeud déterminant. Sur un tel noeud, le mode à priori doit être connu (par lecture dans une table de symbole ou simplement par sa position syntaxique particulière). L'algorithme de liste des modifications est alors appliqué. Si le contexte est faible ou mou, le résultat sera la détermination du mode à postériori. Cette information doit généralement être retransmise en remontant dans l'arbre pour permettre la fin du traitement des noeuds appelants. Cette retransmission est réalisée en appliquant d'abord les macros de liste des modifications, c'est-à-dire quelque chose, soit de la forme :

fort $mode_{post}$ $mode_{pré}$ noeud

soit de la forme :

mou $mode_{pré}$ noeud

Pour les contextes ferme et fort les noeuds sont simplement "transformés" en opérateurs de modification. Mais pour mou et faible cela débouche sur la production de la phrase :

terminé $mode_{post}$

Donc, dans tous les cas, le mode à postériori du sous-noeud est connu ou peut être déterminé.
L'expansion des macros retransmet effectivement les résultats (s'il y en a) au niveau juste supérieur dans l'arbre.

Un noeud de niveau supérieur "reçoit" les résultats grâce à un deuxième appel de macro composé du symbole terminé et du mode résultant. L'action nécessaire est réalisée à ce niveau, puis, les résultats sont transmis au niveau encore supérieur comme précédemment.

Donc, les actions, à réaliser pour chaque mode, peuvent être représentées par un ensemble d'appels de macros et de remplacements. Chaque remplacement positionne l'état pour le prochain appel jusqu'à ce que le traitement de ce noeud soit terminé. Le remplacement final doit être conforme à la grammaire de base, c'est-à-dire être un arbre d'opérateur simple.

Maintenant que les principes de la méthode ont été dégagés, il est utile de voir comment ils s'appliquent à des opérateurs particuliers.

5.231 - Le noeud affectation-

Représenté en code préfixé, l'affectation est composée d'un symbole d'affectation (:=) suivi de deux noeuds, les opérandes gauche et droit respectivement. Une modification est appliquée à une affectation, en préfixant son code pour le marqueur "descente" (∇), la force de contexte et le mode à postériori désiré (s'il est connu). L'appel de macro pour déterminer les modifications est donc :

$$\text{noeud} \rightarrow \text{'}\nabla\text{'}\ \text{contexte}\ \text{mode}_{post}\ \text{':='}\ \text{noeud}_g\ \text{noeud}_d$$

[Afin de rendre l'écriture en macro plus lisible, des parenthèses seront ajoutées pour montrer la structure et les caractères spéciaux ne seront pas écrits entre apostrophes. D'où :

$$\text{noeud} \rightarrow (\nabla\ \text{contexte}\ \text{mode}_{post})\ \text{':='}\ \text{noeud}_g\ \text{noeud}_d \quad]$$

Les modifications à appliquer à l'affectation dépendent du mode de la partie gauche qui devient le mode (à priori) de l'expression toute entière. L'opérande gauche étant en contexte mou, son mode n'est pas encore connu. Donc, le remplacement de macro spécifiera la modification de la partie gauche dans un contexte mou et avec un mode à postériori inconnu.

noeud → (∇ contexte $mode_{post}$) ':=' $noeud_g$ $noeud_d$

= (□ contexte $mode_{post}$) ':=' (∇ mou inconnu) $noeud_g$ $noeud_d$

Au moyen d'une série de transformations qui seront détaillées plus tard la sous-phrase

(∇ mou inconnu)

est transformée en

(terminé mode)

Ce mode est celui déterminé en modifiant la partie gauche et est, donc, celui de l'affectation. Le résultat est reconnu par un nouvel appel de macro :

noeud → (□ contexte $mode_{post}$) ':=' (terminé $mode_g$) $noeud_g$ $noeud_d$

Il reste deux tâches à accomplir :

(1) - démarrer le processus de modifications pour la partie droite,

(2) - finir la détermination pour la phrase elle-même (formant un tout)

Le problème (1) est résolu en prenant le mode gauche, en lui enlevant un rep (comme l'indique la définition) et en considérant ce mode comme le mode à postériori de la partie droite. La sous-phrase :

(∇ fort $mode_d$)

sera finalement transformée en une série d'opérateurs appliquant les modifications à ce noeud.

Le deuxième problème, qui transmet les résultats de la détermination lors de la remontée dans l'arbre, est pris en compte en insérant la sous-phrase :

$$(\Delta \text{ contexte } \text{mode}_{post} \text{ mode}_{g})$$

devant l'ensemble du noeud. Ceci sera finalement transformé en modifications pour le noeud affectation.

La deuxième macro peut maintenant être totalement écrite :

noeud → ($\square$ contexte mode_{post}) ':=' (terminé mode_{g}) noeud_{g} noeud_{d}

= si mode_{g} → 'rep' mode_{d}

alors (Δ contexte mode_{post} mode_{g}) ':=' noeud_{g} (∇ 'fort' mode_{d}) noeud_{d}

fsi

Une fois que cette seconde série de remplacements est faite, le résultat final peut être exprimé dans la grammaire de base c'est-à-dire sous forme préfixée simple.

Il reste une transformation à discuter. Le marqueur "remontée" (Δ) doit être enlevé de façon à pouvoir appliquer les transformations de la liste des modifications, détaillées dans la première partie de ce chapitre (5.22).

noeud → (Δ contexte mode_{post} $\text{mode}_{pré}$)

= si contexte → 'mou' alors 'mou' $\text{mode}_{pré}$

sinsi contexte → 'faible' alors 'faible' $\text{mode}_{pré}$

sinsi contexte → 'ferme' alors 'ferme' mode_{post} $\text{mode}_{pré}$

sinsi contexte → 'fort' alors 'fort' mode_{post} $\text{mode}_{pré}$

fsi

Cette macro est nécessaire pour la seule raison technique que les formats des transformations des différents contextes ne correspondent pas. Une autre solution serait de tester le contexte dans les macros, pour chaque type de noeud : solution beaucoup plus ennuyeuse.

5.232 - Le noeud identificateur -

Ce noeud très simple va être traité afin de permettre la réalisation d'un exemple complet de détermination de modifications. Une "demande", pour modifier un noeud identificateur, en provenance de noeuds situés plus haut dans l'arbre, est reçue sous la forme d'une phrase du type :

(∇ contexte mode) idf

Il suffit de déterminer le mode à priori de l'identificateur (en le cherchant dans la table des symboles) et d'appliquer directement l'algorithme de liste des modifications.

noeud → (∇ contexte $mode_{post}$) idf

= si $mode_{pré}$: a le mode de (idf)

then (Δ contexte $mode_{post}$ $mode_{pré}$) idf

fsi

5.233 - Un exemple complet -

Considérons l'expression simple :

a := b := c ;

avec les déclarations suivantes, pour les identificateurs :

rep ent a = ... ;

proc rep ent b = ... ;

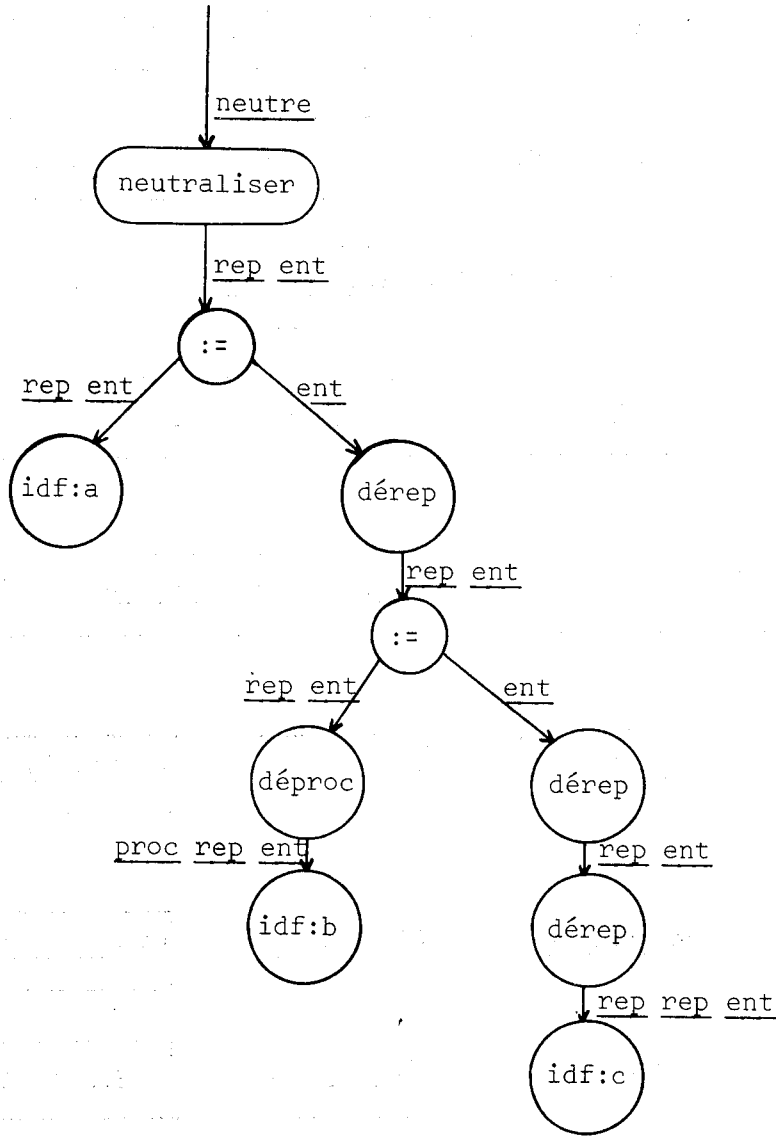
rep rep ent c = ... ;

La forme préfixée de l'expression initiale est :

:= a := b c

Le mode à postériori et le contexte intiaux sont neutre et fort (comme c'est toujours le cas pour une proposition suivie d'un ";").

Le résultat désiré (exprimé sous forme d'arbre) est :



(Les modes correspondant à chaque phrase du développement ont été ajoutés pour servir de guides).

Emcore une fois, les appels de macro vont être représentés par les transformations qu'ils provoquent.

(∇ fort neutre) := a := b c

(□ fort neutre) := (∇ mou inconnu) a := b c

(□ fort neutre) := (Δ mou inconnu ← rep ent) a := b c

(□ fort neutre) := (mou rep ent) a := b c

(□ fort neutre) := (terminé rep ent) a := b c

(Δ fort neutre ← rep ent) := a (∇ fort ent) := b c

neutraliser := a (□ fort ent) := (∇ mou inconnu) b c

neutraliser := a (□ fort ent) := (Δ mou inconnu ← proc rep ent) b c

neutraliser := a (□ fort ent) := (terminé rep ent) déproc b c

neutraliser := a (Δ fort ent ← rep ent) := déproc b (∇ fort ent) c

neutraliser := a dérep := déproc b (Δ fort ent ← rep rep ent) c

neutraliser := a dérep := déproc b dérep dérep c

La dernière ligne est simplement la représentation préfixée de l'arbre dessiné ci-dessus. Tous les pas intermédiaires de l'algorithme de liste des modifications ont été omis dans le diagramme ci-dessus.

5.234 - Equilibrage -

Maintenant que la méthode pour prendre en charge les modifications a été établie, il serait intéressant de voir comment elle s'applique à un problème plus nouveau, qui rend invalide, pour la détermination des modifications, beaucoup de démarches directes parmi les plus simples.

L'équilibrage est une notion introduite en ALGOL 68 pour expliquer comment la force du contexte d'une proposition peut être transmise à ses propositions constituantes, de différentes manières. Considérons comme exemple la proposition si. Une de ses propositions constituantes (la partie alors ou la partie sinon) a le même contexte qu'elle même. L'autre partie du choix a le contexte fort. On peut, donc, utiliser deux manières distinctes, pour arriver aux modifications correctes à appliquer aux différentes parties de la proposition si. Si certaines modifications demandent un contexte bien précis pour être applicable, une seule de ces manières réussira. (Si les deux manières réussissent, elles doivent, bien sûr, fournirent des résultats identiques). L'équilibrage est le nom donné à ce processus servant à trouver le choix correct de transmission du contexte de façon à ce que toutes les modifications soient déterminables.

Soit, comme exemple, la phrase :

z := z2 + si p alors r sinon z3 fsi

où

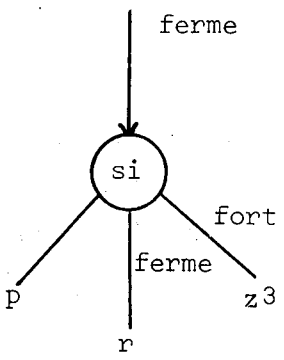
z, z2, z3 sont déclarés compl,

r est déclaré réel,

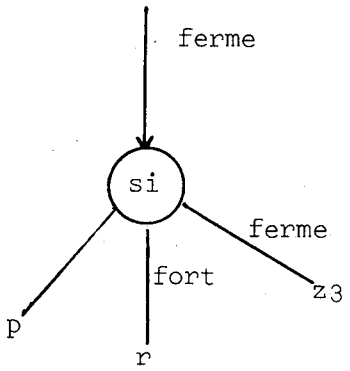
et p est déclaré bool.

La proposition si est en contexte ferme et doit avoir le mode compl, car la proposition est une opérande de l'opérateur "+" entre complexes. Ceci veut dire que l'équilibrage permet deux possibilités :

(a)



(b)



Si (a) est essayée, cela ne marchera pas car r (réel) ne peut pas être élargi en compl dans un contexte ferme. Par contre, (b) marchera car la modification élargir est possible en contexte fort et car aucune modification n'est nécessaire pour faire de Z3 un objet compl.

Plus généralement, l'"esprit" de la méthode consiste à transmettre des contextes à l'intérieur de la proposition entraînant l'équilibrage, en essayant chaque possibilité en parallèle et en créant peut-être encore plus d'équilibrages au cours du processus (propositions cas, acheveurs) jusqu'à ce qu'un ou plusieurs chemins réussissent à déterminer les modifications pour toute la phrase.

Ceci ressemble fortement à la technique utilisée, dans la première partie de ce chapitre, pour décider quelles modifications appliquer quand il semble y en avoir plusieurs possibles. Tous les développements possibles étaient essayés, en s'appuyant sur l'hypothèse (valide) que l'essai d'un développement incorrect conduisait à une analyse incapable de se terminer.

De même, les macros utilisées, ci-dessous, pour décrire la modification de la proposition si ont exactement cette forme.

5.2341 - UNE DESCRIPTION PAR MACRO DES MODIFICATIONS DE LA PROPOSITION CONDITIONNELLE -

Le traitement du noeud si dépend assez fortement de son contexte. Il doit, donc, être testé dans la macro afin de pouvoir faire les remplacements appropriés. Ceci sera discuté une fois la macro présentée. Pour gagner de la place, les noms du noeud, du contexte et du mode sont représentés par n, c et m, respectivement. Les indices p, a et s se réfèrent au prédicat, à la partie alors et à la partie sinon.

```
(1)  n → (∇ c m) 'si' n_p n_a n_s
(2)     = si c → 'fort'
(3)       alors 'si' (∇ 'fort bool')n_p (∇ 'fort' m)n_a (∇ 'fort' m)n_s
(4)       sinsi c → 'ferme'
(5)       alors 'si' (∇ 'fort bool')n_p (∇ 'ferme' m)n_a (∇ 'fort' m)n_s
(6)       ou    'si' (∇ 'fort bool')n_p (∇ 'fort' m)n_a (∇ 'ferme' m)n_s
(7)       sinsi c → 'faible'
(8)       alors 'si' (∇ 'fort bool')n_p (∇ 'faible' m)n_a n_s
(9)       ou    'si' (∇'fort bool')n_p n_a (∇ 'faible' m)n_s
(10)      sinsi c → 'mou'
(11)      alors 'si' (∇ 'fort bool')n_p (∇ 'mou' m)n_a n_s
(12)      ou    'si' (∇ 'fort bool')n_p n_a (∇ 'mou' m) n_s
(13)      fsi
```

La macro est appelée, comme d'habitude, avec un certain contexte et un certain mode à postériori. Le choix des remplacements est déterminé en fonction de la force du contexte (fort (2), ferme (4), faible (7) ou mou (10)).

Dans tous ces remplacements le noeud prédicat est toujours modifié en contexte fort vers le mode à postériori bool.

Cette détermination de modifications est complètement indépendante du reste du processus. Considérons, individuellement, chaque cas de contexte.

Remarque :

Le mode à postériori est simplement transféré aux parties alors et sinon sans application de l'algorithme de liste des modifications.

Fort : l'équilibrage est inopportun, car la seule action à faire est d'appliquer le mode en contexte fort aux parties alors et sinon.

Ferme : Il y a deux remplacements possibles (5 et 6) qui sont, donc, tentés en parallèle. Si la proposition conditionnelle est correctement écrite, au moins une des deux analyses tentées réussira.

Faible et mou : comme avec le contexte ferme, deux analyses sont tentées en parallèle. Ici, cependant, le mode à postériori n'est pas connu (le mode dans l'appel (1) est simplement le mode inconnu). Donc l'autre sous-noeud ne peut pas simultanément être modifié en contexte fort, mais doit attendre jusqu'à ce que la sous-phrase

$$\underline{\text{terminé}}\ \text{mode}_{\text{post}}$$

apparaisse (lorsque les séries usuelles de transformations ont été appliquées au noeud mou ou faible). Cela signifie qu'il doit y avoir un deuxième appel de macro pour le noeud si, complétant la détection de la partie en contexte fort et transmettant ensuite le résultat à un noeud supérieur dans l'arbre (qui attend lui aussi "terminé mode") :

$$n \rightarrow \text{'}\underline{\text{si}}\text{'}\ n_p\ (\underline{\text{terminé}}\ m)n_a\ n_s$$
$$= (\underline{\text{terminé}}\ m)\ \text{'si'}\ n_p\ n_a\ \nabla\ (\underline{\text{fort}}\ m)n_s$$

$$n \rightarrow \text{'si'}\ n_p\ n_a\ (\underline{\text{terminé}}\ m)n_s$$
$$= (\underline{\text{terminé}}\ m)\ \text{'}\underline{\text{si}}\text{'}\ n_p\ \nabla\ (\text{fort}\ m)n_a\ n_s$$

5.235 - Autres noeuds ALGOL 68 -

La même méthode, de description des opérations nécessaires pour la détermination des modifications, peut être également utilisée pour tous les autres noeuds ALGOL 68. Bien sûr, chaque type de noeud à ses propres particularités, mais, une fois les caractérisitiques du noeud assimilées, la technique de définition par macros-syntaxiques ne présente aucune difficulté spéciale, à une exception près dont nous reparlerons plus tard. Le travail, pour chacun des noeuds, a été décrit en ALGOL 68 dans les dossiers du compilateur de Grenoble aussi bien pour la version initiale du Rapport que pour la version révisée.

5.24 - Identification d'opérateur -

Afin d'évaluer les macros-syntaxiques en tant que technique de définition, il est nécessaire d'en montrer aussi bien les faiblesses que les avantages. A ma connaissance, le seul cas défavorable est la détermination des modifications pour un opérateur ALGOL 68.

Il est nécessaire d'entreprendre simultanément deux opérations distinctes :

(1) - déterminer les modifications pour les opérandes

(2) - identifier l'opérateur

Les opérateurs doivent être identifiés car le même symbole, par exemple : plus (+), peut posséder comme valeurs, plusieurs routines différentes.

La routine particulière à appliquer dépend du mode du (des) opérande(s). Dans le cas de l'opérateur dyadique plus (+), le prologue standard fournit :

(1) (ent, ent) ent

(2) (réel, réel) réel

(3) (chaîne, chaîne) chaîne

(4) (car, car) chaîne

De plus, l'utilisateur peut définir d'autres routines plus. Malheureusement, ce sont les déclarations qui fournissent les modes à postériori des opérandes. Cela signifie que les opérandes sont à modifier vers un mode qui dépend de la déclaration particulière de l'opérateur, que l'on identifie.

Etant donné que l'identification de l'opérateur dépend des modes des opérandes modifiés, et que les modifications sur les opérandes dépendent de l'opérateur identifié, comment les deux peuvent-ils être déterminés ? Une solution préliminaire est d'utiliser la méthode classique consistant à essayer, en parallèle, l'identification de tous les opérateurs possibles. Si les déclarations d'opérateur appartiennent toutes à la même région, seule une des analyses tentée doit réussir.

Par exemple, si nous avons la phrase (préfixée) :

(∇ fort réel) + re pe

il faut, alors, essayer les transformations suivantes :

(1) (Δ fort réel ← ent) $+_1$ (∇ ferme ent) re(∇ ferme ent) pe

(2) (Δ fort réel ← réel) $+_2$ (∇ ferme réel) re (∇ ferme réel) pe

(3) (Δ fort réel ← chaîne) $+_3$ (∇ ferme chaîne) re (∇ ferme chaîne) pe

(4) (Δ fort réel ← chaîne) $+_4$ (∇ ferme car) re (∇ ferme car) pe

... etc ....

Si les modes des identificateurs re et pe sont respectivement rep ent et proc ent, seule la première transformation se terminera avec succès :

élargir $+_1$ dérep re déproc pe

La difficulté avec cette approche est que toutes les déclarations n'appartiennent pas forcément à la même région. Dans ce cas, il serait possible d'identifier plus d'un exemplaire du même opérateur, à la condition qu'il n'y en est qu'un seul dans la région la plus interne. En accord avec la définition d'ALGOL 68, cette dernière identification est valide. Si les transformations ont été lancées en parallèle, il n'existe

aucun moyen d'en arrêter une série, uniquement parce qu'une autre a marché. Le problème semble, donc, être celui du parallélisme.

Si les identifications possibles d'un opérateur sont essayées en série - c'est-à-dire l'une après l'autre, en partant de la région la plus interne - il suffit alors de choisir la première identification qui marche. C'est, en fait, ce qui est fait dans le compilateur de Grenoble.

Les essais en série sont moins faciles à définir avec les macros-syntaxiques.

Une approche peut être décrite de la manière suivante :

Supposons que pour une macro, on désire faire le remplacement R1 mais s'il ne marche pas, on veut faire le remplacement R2.
Construire une phrase de remplacement "non R1" telle qu'on obtienne (après transformation) la phrase "transformé non R1" si et seulement si R1 n'analyse pas. Les deux macros suivantes satisfont à ces demandes :

M → C = R_1 ou non R1

M → transformé non R1 = R2

La méthode peut être trivialement étendue à plus de deux remplacements. Pour le problème particulier posé - à savoir, la détermination des modifications - cela n'est pas d'une très grande utilité. La manière, de construire une phrase qui analysera si et seulement si un opérateur est non identifié, n'est pas immédiate.

Une autre approche offre plus de promesses mais elle nécessite un changement dans le mécanisme syntaxique. Essentiellement, cela consiste à ajouter un "point d'arrêt" analogues à ceux utilisé dans les systèmes d'exploitation. Supposons que le premier remplacement R1 soit préfixé du marqueur spécial pointarret :

M → C = pointarret R1

Si R1 analyse, alors, le marqueur spécial est simplement ignoré (il ne prend pas part à l'analyse). Si R1 ne marche pas, l'état de l'analyse

lorsque le dernier point d'arrêt a été appliqué, est restauré. Cette restauration s'accompagne de la mise en place du marqueur spécial "échec". Le deuxième remplacement peut, alors, être essayé :

$$M \rightarrow \text{'échec'} = R_2$$

Si des indices sont ajoutés aux points d'arrêt, cette méthode peut être étendue à un nombre quelconque d'essais en série.

Il est important de s'apercevoir que cette méthode ne peut pas être simulée dans le cas général en ayant simplement la dernière des séries de transformations prise en charge par une macro qui ajouterait explicitement le marqueur échec quand toutes les autres ne marchent pas.

Il y a deux raisons :

Premièrement, la phrase courante de macro, ne coïncide pas forcément avec le remplacement original (c'est le cas pour la détermination des modifications).

Deuxièmement, il n'y a peut-être pas, toujours, une manière d'écrire un (ou beaucoup) appel de macro qui couvre tous les cas possibles d'échec (ceci est analogue à construire "nonR").

Bien que la méthode du point d'arrêt résoudrait certainement le problème d'identification des opérateurs en ALGOL 68, il s'agit simplement d'une idée qui n'a pas été testée. Par ailleurs, il n'est pas du tout évident qu'il n'y ait pas de difficultés dans l'implémentation de la sauvegarde de l'état de l'analyse.

5.25 - Evaluation de l'application des macros-syntaxiques au problème des modifications -

La raison, de l'application si poussée des macros-syntaxiques à la description de la partie "modification" de notre compilateur, était de tester leur facilité d'emploi dans une situation pratique. Comme déjà mentionné précédemment, les macros sont prévues pour un double but : la définition et l'implémentation.

Tant qu'il ne s'agit que de définition, cette méthode semble tout à fait digne d'intérêt. Avec l'exception possible de l'identification des opérateurs, les macros-syntaxiques semblent être capables de définir toutes les constructions et les résultats nécessaires. La définition me semble, personnellement, être plus lisible que la grammaire de Van Wijngaarden considéré souvent comme quelque peu obscure. Une question pertinente serait peut-être de savoir s'il existe une quelconque définition des modifications ALGOL 68 qui soit lisible. Ces macros semblent être raisonnablement concises, sans trop de contorsions, même s'il existe des cas ou un certain nombre de factorisations seraient utiles (se rappeler le cas de la proposition conditionnelle où le prédicat booléen en contexte fort a été répété dans chaque remplacement possible).

Il est clair que le choix d'une forme intermédiaire polonaise préfixée a considérablement facilité le problème de définition de macro. L'existence d'un seul symbole non-terminal significatif permet l'écriture de macro-syntaxiques correspondant, de manière très proche, aux grammaires de Chomsky de type 1, ce qui est une forme très pratique. Cette connexion et ses conséquences sont discutées plus à fond dans la section 1.7.

Il y a une dépendance assez lourde avec la notion de séries de transformations en parallèle où celles qui réussissent doivent toutes fournir des résultats identiques. Pour des besoins de définition cela est très pratique, mais cela entraîne la perte d'une caractéristique importante d'implémentation : cela donne la possibilité de répéter plusieurs fois un travail qui a déjà conduit à un échec. C'est une des raisons pour lesquelles les analyseurs classiques à retour-arrière non limités sont non linéaires. De plus, la méthode en parallèle persiste à chercher des solutions mauvaises, même lorsque la bonne a été trouvée, ce qui n'est peut-être pas un désavantage si on prend en considération la détection des erreurs.

Dans le cadre de cet essai, la phase de définition est très favorisée par rapport à la phase d'implémentation. Il y a, bien sûr, beaucoup de possibilités de rétablir l'équilibre, une fois établie la politique d'écriture des macros. Il n'est pas difficile de trouver des cas ou une macro peut être réécrite, d'une manière, sans doute, plus compliquée et moins lisible, mais conduisant à l'exécution d'un nombre plus petit de remplacements.

6 - TECHNIQUES DE GENERATION EN ALGOL 68 -

Dans ce chapitre, est traitée la dernière étape de toute compilation, c'est-à-dire la production du code machine effectif. Afin d'aller au-delà des généralisations habituelles, je présenterai un exemple spécifique provenant de mes travaux sur la génération de code pour le compilateur ALGOL 68 de Grenoble implémenté sur une machine IBM 360/67.

Supposons que l'arbre d'opérateurs a été produit et que tous les opérateurs de modification lui ont été ajoutés, comme expliqué dans les chapitres précédents. A chaque type de noeud opérateur est associée une routine qui est supposée générer le code nécessaire. Il est important de réaliser que le code à générer dépend du contexte dans lequel l'opérateur est trouvé. Par exemple, si la partie droite d'une affectation est une expression (non dégénérée), le code pour la partie droite fournira la valeur de l'expression dans un registre central. Le code pour l'affectation consiste simplement à ranger ce registre dans la cellule de mémoire adressée par la partie gauche. Si, par contre, la partie droite est simplement un identificateur, il faudra alors pour l'affectation générer d'abord une instruction de chargement du registre central.

6.1. - Définition de l'attribut de position -

Pour éviter de se perdre dans les détails, la notion de dépendance vis-à-vis du contexte doit être systématisée. La partie la plus importante du problème est de connaître où les valeurs de chaque opérande et celle du résultat sont à trouver et comment elles sont à référencer. C'est ce que j'appellerai l'attribut de position d'une valeur.

La valeur peut être trouvée en utilisant le "champ R" d'une instruction (par exemple quand la valeur est déjà dans un registre central) ou les champs base et déplacement (par exemple quand la valeur est encore en mémoire centrale). C'est la partie "type du champ instruction" de l'attribut de position.

Bien sûr, pour trouver effectivement la valeur, il faut aussi connaître le numéro de registre (pour le champ R) ou le numéro de registre de base et la valeur du déplacement (pour le champ adresse). C'est la partie "emplacement" de l'attribut de position.

La quantité désignée par le registre ou le champ adresse peut représenter soit la valeur littérale désirée soit l'adresse machine de cette valeur. Ceci correspond à la partie "niveau d'indirection" de l'attribut de position.

Considérons maintenant une définition plus rigoureuse de "position". Dans les instructions en assembleur 360 ci-dessous, le symbole "valeuremp" représentera la valeur emplacement et le symbole "X" n'importe quel registre central. Les quatres combinaisons possibles de type d'instruction et de niveau d'indirection seront désignés par L, M, R et A comme suit :

	type champ instruction	niveau d'indirection
L (littéral dans champ instruction)	base + déplacement	direct
M (valeur en mémoire)	base + déplacement	indirect
R (valeur dans un registre)	registre	direct
A (adresse de valeur dans un registre)	registre	indirect

Un opérande est dit de :

Type L si et seulement si la valeur est dans un registre X après exécution de LA X, valeuremp

Type M si et seulement si la valeur est dans un registre X après exécution de L X, valeuremp

Type R si et seulement si la valeur est dans un registre X après exécution de LR X, valeuremp

Type A si et seulement si la valeur est dans un registre X après exécution de L X, 0(valeuremp)

6.2 - Attributs de position dans la table des symboles -

Un noeud identificateur dans l'arbre d'opérateurs aura comme paramètre un pointeur vers l'identification du symbole utilisé. A cause d'une optimisation absolument nécessaire pour les déclarations "contractées" (c'est-à-dire :

ent a ; pour rep ent a = loc ent;)

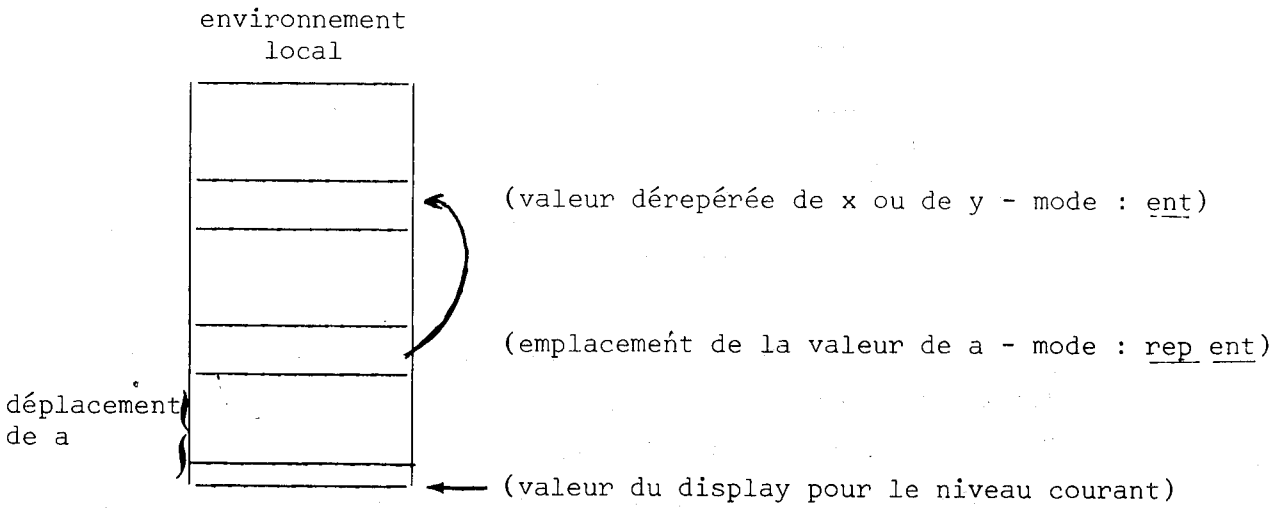
deux types d'entrées dans la table des symboles doivent être distinguées.

Considérons d'abord le cas non optimisé.

Si la déclaration est :

rep ent a = si p alors x sinon y fsi

de la place est alors réservée dans l'environnement local pour la valeur "a" ainsi que pour le résultat de l'expression (p | x| y)

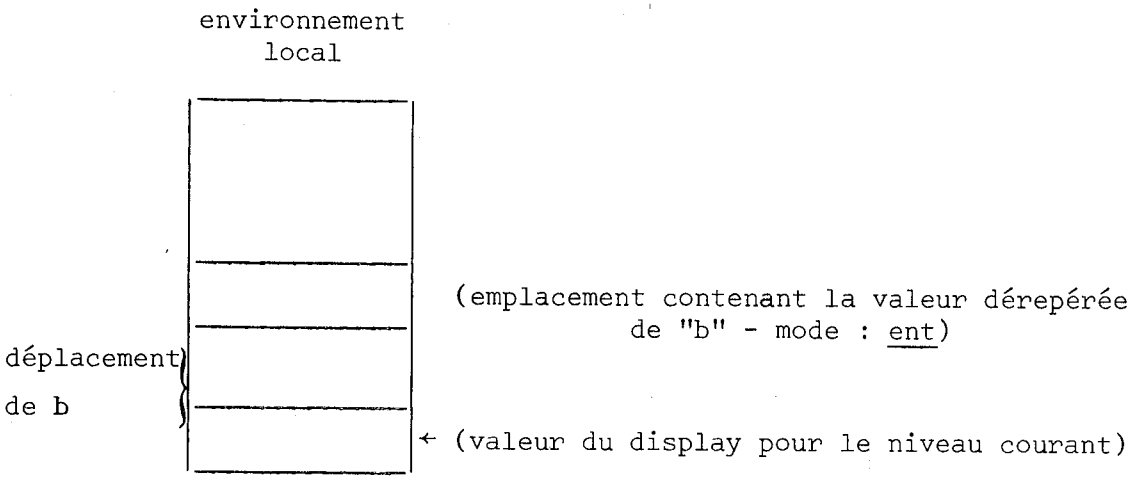


Le déplacement de a et son numéro de niveau sont rangés dans la table des symboles. Il faut noter que ceci sert à localiser une valeur de mode rep ent et non pas uniquement les contenus d'une cellule de mémoire de ce nom, ce qui serait le cas dans la plupart des autres langages. Cette valeur n'est pas statiquement connue car elle peut varier d'une exécution à l'autre suivant les résultats de "p". Elle doit donc être trouvée dans la cellule de mémoire adressée par la base et le déplacement rangés dans l'entrée "a" de la table des symboles. Cela signifie que cette adresse a un attribut de position M. La valeur, ici de mode rep ent, est trouvée en exécutant une instruction de chargement utilisant cette adresse.

Considérons, d'autre part,

rep ent b = loc ent

Il s'agit d'un cas où la valeur de "b" (encore de mode rep ent) est statiquement connue. La valeur est l'adresse retournée par le générateur local qui peut occuper toujours la même position dans l'environnement local. Si la valeur de "b" est connue du compilateur, il n'y a pas besoin de réserver de la place pour elle. La seule réservation nécessaire est pour la valeur résultant de l'application de la modification dérepérer à la valeur "b".



Dans ce cas, la paire (base - déplacement), rangée dans la table des symboles, ne localise pas mais est, littéralement, la valeur de "b". Dans ce cas, l'attribut de position est donc L car on peut obtenir la valeur de b en exécutant une instruction de chargement d'adresse (LA).

La distinction essentielle entre ces deux cas est le caractère statique (L) ou dynamique (M) de la valeur possédée par le nom externe.

Les cas statiques courants sont illustrés par :

```
ent s ;
rep ent s = loc ent ;
rep ent t = s ; (s a le même mode que t)
```

Tous les autres peuvent être traités comme étant dynamique et sont donc de type M. Cela inclut :

- les générateurs globaux (rep ent d = tas ent)
- les expressions non dégénérées (rep ent d = p (a))
- les identificateurs modifiés (ent d = s)
- les paramètres formels (proc p = (ent d) neutre : ...)

6.3 - Les attributs de position pour les résultats des opérateurs de l'arbre -

L'opérateur de chaque noeud implique de la génération de code pour son (ses) opérande(s), d'abord, en appelant leur routine de génération associées, puis, en générant directement du code en accord avec les attributs de position retournés par cette (ces) routines(s). Tout ceci sera plus clair après avoir examiné l'interaction entre deux routines simples de génération : celle pour les noeuds identificateurs et celle pour la modification dérepérer.

La routine pour l'identificateur est la plus simple : elle ne génère aucun code et se contente de retourner l'attribut de position trouvé dans la table des symboles.

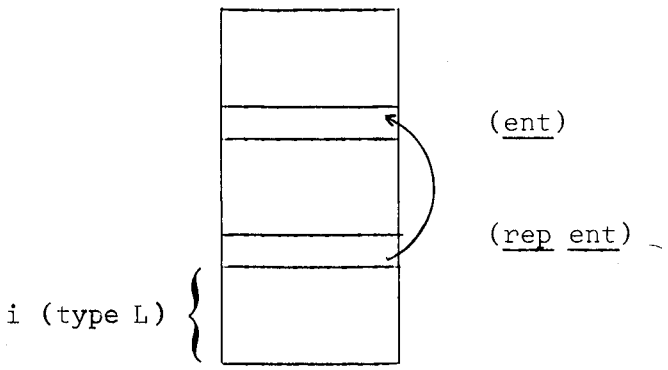
La routine pour la modification dérepérer appelle son opérande, ce qui entraîne sa génération et la restitution de son attribut de position. Puis, en accord avec cet attribut, elle génère une ou zéro instruction et retourne son attribut de position :

Position de l'opérande	Génération	Position finale
L, valeuremp	-	M, valeuremp
M, valeuremp	L X, valeuremp	A, X
R, valeuremp	-	A, valeuremp
A, valeuremp	L X, 0(valeuremp)	A, X

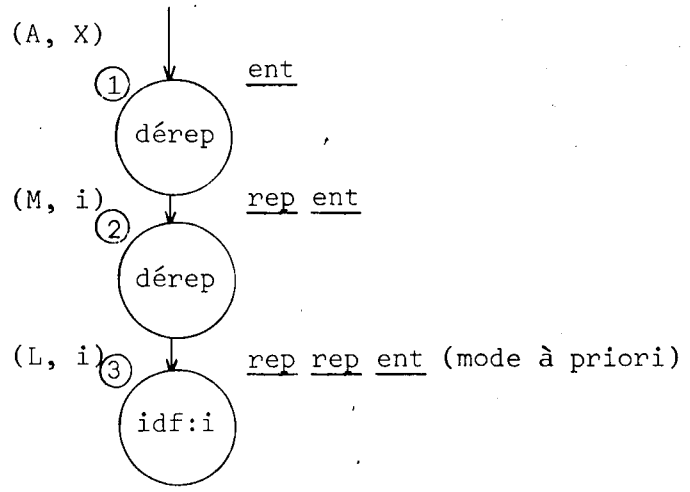
Remarquons que dans deux cas (opérandes L et R) aucun code n'est généré. Il suffit de considérer la valeur emplacement comme une adresse et non plus comme un littéral. Ceci est reflété par le changement de l'attribut de position pour le résultat (L → M et R → A).

Considérons un exemple utilisant ces deux types de noeud :

i est déclaré par : rep rep ent i = loc rep ent et est représenté en mémoire par :



le forceur ent(i) entraîne la génération du segment d'arbre d'opérateurs :



Le noeud ① appelle le noeud ② qui à son tour appelle le noeud ③, c'est-à-dire le noeud identificateur.

Ce noeud ne génère aucun code et retourne la position de "i" (L, i). Le noeud (2), en accord avec l'attribut L de son opérande, ne génère rien et retourne la position (M, i). Le noeud (1) remarque que son opérande a un attribut M et doit donc générer

```
L X, i
```

et retourner la position (A, X). Cela signifie que la valeur entière est accessible en utilisant l'adresse trouvée dans le registre "X".

6.4 - Génération de l'affectation en utilisant les attributs de position -

Un cas plus intéressant pour les attributs de position est celui de l'opérateur affectation. Ici il y a deux opérandes, chacun d'eux pouvant avoir un type quelconque de position. Il semblerait donc qu'il y ait seize générations distinctes possibles. Cela est vrai, en principe, mais la routine de génération peut être factorisée en remarquant que le code généré pour obtenir la valeur de l'opérande droit est complétement indépendant de la manière dont est utilisée la position de la valeur en partie gauche, pour le stockage.

Donc, après avoir généré les deux opérandes et déterminé leurs deux positions, le code pour charger la valeur de la partie droite peut être généré :

position de l'opérande droit	génération
(L, valeuremp$_d$)	LA X, valeuremp$_d$
(M, valeuremp$_d$)	L X, valeuremp$_d$
(R, valeuremp$_d$)	(X étant valeuremp$_d$)
(A, valeuremp$_d$)	L X, O(valeuremp$_d$)

La valeur de la partie gauche peut ensuite être obtenue et utilisée pour ranger la valeur qui est dans le registre X. La valeur de la partie gauche est nécessairement une adresse machine (elle est de mode rep ....) et la destination de la valeur de la partie droite est l'emplacement pointé par cette adresse.

Donc quand la valeur de la partie gauche a la position L (littéral), elle est sous une forme correcte pour être utilisée dans une instruction de rangement. Si elle est, par contre, dans la position M (la valeur est localisée par une adresse), cette valeur doit être chargée dans un registre avant de pouvoir être utilisée comme une adresse de rangement.

Position opérande gauche	Génération	Position du résultat
(L, valeuremp$_g$)	ST X, valeuremp$_g$	(L, valeuremp$_g$)
(M, valeuremp$_g$)	L X2, valeuremp$_g$	(R, X2)
	ST X, 0(X2)	
(R, valeuremp$_g$)	ST X, 0(valeuremp$_g$)	(R, valeuremp$_g$)
(A, valeuremp$_g$)	L X2, 0(valeuremp$_g$)	(R, X2)
	ST X, 0(X2)	

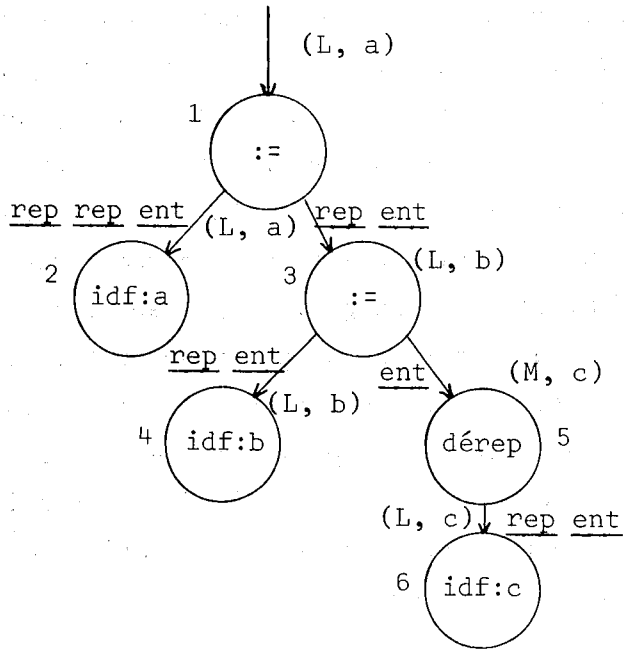
Illustrons ceci par un exemple simple. Le texte ALGOL 68 est :

```
a := b := c
```

avec les déclarations pour a, b, et c :

```
rep ent a, ent b, c ;
```

Après traitement, les affectations fournissent l'arbre d'opérateurs ci-dessous :



Les routines de génération passent de noeud en noeud et produisent la sortie suivante dans laquelle il faut remarquer comment la position de chaque noeud résultat est mise à jour.

noeud	Génération	Position
idf_2	-	(L, a)
idf_4	-	(L, b)
idf_6	-	(L, c)
$dérep_5$	-	(M, c)
$:=_3$	L X, c	(L, b)
	ST X, b	
$:=_1$	LA X, b	(L, a)
	ST X, a	

6.5 - Avantages et inconvénients des attributs de position -

Bien que, dans leurs fondements les positions d'attributs soient une formalisation d'une stratégie classique de génération (cf. GRIES [12], Mc KEEMAN [17], ... etc...), leur utilisation pour ALGOL 68 prend une importance accrue par la nature très systématique du langage. Par exemple, la plutôt mystérieuse modification "dérepérer" conduirait a des inefficacités importantes si elle était implémentée d'une manière collant à la définition formelle du langage : la plupart des variables devraient être localisées indirectement par un pointeur dont la valeur indirecte serait à charger pour toute référence à chacune de ces variables. Le mérite des attributs de position est de permettre la production de code "normal" dans le cas ou le programmeur a écrit des expressions et des affectations "normales" et de permettre aussi la production de code correct dans les cas plus sophistiqués. Par exemple :

```
a := b + c * d
```

produira exactement le code que chacun pense même si b, c, et d de mode rep réel sont à dérepérer.

Le fait qu'il y ait quatre types de base est lié à la nature d'ALGOL 68. En PL/1, par exemple, il suffirait probablement d'avoir les types M et R (les contenus d'une valeur en mémoire et la valeur dans un registre central).

D'autre part, les attributs de position ne représentent qu'une première forme, absolument nécessaire, d'optimisation locale. Rien n'est prévu pour garder la trace des contenus de chaque registre central. Si l'exemple de la double affectation ci-dessus avait été généré pour des identificateurs ayant tous le même mode (par exemple rep ent) le code de sortie aurait été :

```
L   X, c
ST  X, b
L   X, b
ST  X, a
```

dans lequel le deuxième chargement est manifestement superflu. La difficulté dans cet exemple particulier est que le résultat de la première affectation (de mode rep ent et de type L) est dérepéré et donne une adresse de type M. En faisant ainsi, la procédure "oublie" que la valeur dérepérée est déjà dans un registre central (X) et qu'il n'y a donc besoin d'aucune instruction pour la charger.

Comme la plupart des problèmes de ce genre, celui-ci peut être résolu en étendant les attributs de position pour y inclure aussi bien de l'information sur la valeur obtenue en dérepérant la valeur courante, que la valeur elle-même.

Bien sûr, il n'est pas difficile d'imaginer des exemples qui ferait échouer cette procédure. Le problème essentiel auquel l'implémenteur se trouve confronté est de trouver un équilibre entre la complexité de sa solution et la qualité du code produit. C'est une question d'objectifs.

6.6 - Autres attributs de génération -

Il ne faudrait pas conclure que toutes les extensions des attributs de position ne sont qu'une question d'optimisation. Les exemples traités jusqu'à présent étaient limités à un seul mode de base (ent) et a une seule construction de base (rep). En fait, pour être réaliste, il serait nécessaire de considérer la forme de tous les objets du langage (dans le cas présent, ALGOL 68) et l'organisation hardware de la machine (dans le cas présent, IBM 360).

Aussi loin que la machine est concernée, les attributs doivent correspondre fonctionnellement aux champs des instructions machines à générer. L'attribut emplacement, qui spécifie les valeurs des champs base et déplacement, peut être étendu pour contenir une valeur de champ index lorsqu'il est effectivement utilisé.

Une extension moins évidente concerne les quatre éléments binaires du code condition de l'IBM 360. Le résultat d'un opérateur de relation ALGOL 68 sera une valeur booléenne codée dans ces quatre éléments binaires. La valeur de l'attribut correspondant à un modèle binaire particulier peut être utilisée pour décider ce que doit être le champ condition de l'instruction conditionnelle de branchement à générer.

Comme les registres de la machine sont aussi différenciés en fonction de l'arithmétique hardware permise (c'est-à-dire virgule flottante ou virgule fixe), l'allocation des registres centraux doit être fonction du mode ALGOL 68 en question (réel ou ent).

Plus généralement, chaque mode de base ALGOL 68 doit être reflété dans les attributs de génération. Ceux-ci se divisent naturellement en classes où l'aspect le plus important est généralement la taille de l'objet. Bien sûr, les modes composés (structures, unions et tableaux) ont des tailles ne correspondant pas à celles des registres standards et donc, des instructions de déplacement de bloc doivent être utilisées.

6.7 - Sommaire -

Le but de ce chapitre est d'essayer de systématiser certains aspects de la génération de code dépendante de la machine. Très souvent, ceci est considéré comme contradictoire car, étant donné que la génération de code est orientée pour une machine particulière, comment est-il possible d'énoncer quoique ce soit de systématique ? En réalité, les types de calculateurs se divisent, assez naturellement, en classes. En conséquence le même schéma de génération peut être utilisé, avec de légères modifications, pour tous les membres d'une même classe. Ces "légères modifications" sont généralement du genre : combien de registres de base de tel et tel type sont disponibles ... etc.... Tout programmeur expérimenté maniera ces valeurs sous forme de paramètres du programme. Les attributs de génération décrits ci-dessus sont en réalité une implémentation de ce procédé de paramétrisation.

Il n'est pas pensable de demander que ces attributs puissent permettre une implémentation instantanée sur toutes les machines d'une même classe. Ils fournissent, en fait, un moyen d'arriver à une nouvelle implémentation, en ne faisant que les changements minimaux à un programme écrit pour une machine similaire. Beaucoup de ces changements auront une nature semblable à celle des macros, par exemple : les constructions effectives de code, l'adressage absolu, ... etc .... La définition d'un langage de macro pour un tel travail était un des problèmes qui m'avaient été suggérés. Un tel langage, plus concerné par les aspects architecturaux des machines que par les primitives algorithmiques, serait, évidemment, de plus bas niveau que le langage de base décrit dans cette thèse.

Quoiqu'il en soit, les commentaires faits dans ce chapitre ne servent qu'à mettre en lumière une approche d'implémentation de la génération de code, et ne peuvent pas être considérés comme une description complète(par suite d'une quelconque extension de l'imagination) des problèmes invoqués. Cette description pourrait à elle seule former le contenu d'une thèse plutôt volumineuse. La plupart de ce qui est présenté est en fait, le résultat de l'étude de ce problème pour le compilateur ALGOL 68 de Grenoble.

Ici encore, l'influence linguistique d'ALGOL 68 a été très bénéfique car sa nature hautement structurée suggère une forme intermédiaire sous forme d'arbre qui, en définitive, conduit à associer les attributs aux noueds de l'arbre. Cela ne veut pas dire que les langages comme FORTRAN ou PL/1 ne puissent pas bénéficier du même traitement mais plutôt que la méthode est, pour eux, moins évidente et que, peut être, elle ne conduit pas à une exploitation aussi complète.

7 - UNE AMELIORATION AU MECANISME DES MACROS-SYNTAXIQUES -

7.1 - Le problème -

L'une des difficultés qu'on rencontre souvent à l'écriture de macros-syntaxiques pour effectuer quelque traduction est qu'il n'y a pas de place pour "ranger" les phrases de remplacement. On entend par là que toute traduction étant un processus effectué pas-à-pas, les remplacements doivent apparaître temporairement au milieu de la construction partiellement traduite. Ces remplacements doivent être conformes à la syntaxe du langage original ou bien être "trainés" en une forme intermédiaire à examiner par la suite.

La traduction de la forme infixée en BL/1 examinée au paragraphe 3.21 est un bon exemple de ce problème. Le piège a été soigneusement évité en écrivant toujours la partie du langage de base à droite de chaque phrase de remplacement de manière à l'exclure automatiquement du prochain appel qui continuera la traduction partielle.

L'appel de macro, par exemple, dans la méthode directe :

```
A = B * (C + D) * E ;
```

conduit au nouvel appel :

```
t = B * (C + D) ;
```

suivi par la phrase en langage de base :

```
mult t, E → A ;
```

Et puisque le nouvel appel n'inclut aucune forme du langage de base, il peut être traité normalement. La macro s'applique au niveau syntaxique <instruction>. Comme on l'a vu clairement au paragraphe 3.21, c'est un niveau mal commode à utiliser. Il serait bien plus naturel de traduire au niveau de la phrase primitive. Par exemple, on aimerait écrire :

$$\text{expr} \rightarrow \text{expr}_2 \text{ '+' terme}$$

$$= \underline{\text{si}} \ (\text{expr}_2 \overset{*}{\rightarrow} \text{idf}_g) \wedge (\text{terme} \overset{*}{\rightarrow} \text{idf}_d)$$

$$\underline{\text{alorsi}} \ \text{idf}_t : \text{NOUVEAU}$$

$$\underline{\text{alors}} \ \text{idf}_t \ \text{'}\underline{\text{add}}\text{'} \ \text{idf}_g \ \text{','} \ \text{idf}_d \ \text{'}\rightarrow\text{'} \ \text{idf}_t$$

$$\underline{\text{fsi}}$$

(En d'autres termes si une expression était faite d'un ensemble d'identificateurs simples alors un nouvel identificateur aurait été créé et une traduction BL/1 produite). Des macros similaires auraient été nécessaires pour d'uatres phrases primitives. La difficulté apparaît à l'affectation :

$$A = B * (C + D) * E ;$$

La première application de la macro produirait :

$$A = B * (t \ \underline{\text{add}} \ C, D \rightarrow t) * E ;$$

qui termine la traduction puisque l'instruction BL/1 est plongée dans l'expression infixée et n'est pas conforme syntaxiquement.

7.2 - Une solution -

Plutôt que d'éviter ce problème par une astuce qui peut entraîner d'autres difficultés lors de l'écriture de macros syntaxiques, il aurait été préférable d'y faire face, c'est-à-dire de donner un autre "endroit" pour ranger les remplacements.

Supposons que cette "place" soit la chaîne d'entrée d'un nouvel analyseur qui définirait seulement le langage d'arrivée (avec une nouvelle grammaire). La phrase transférée disparaitrait complètement de l'analyseur primitif et par conséquent la grammaire primitive ne la définirait pas.

Cette idée généralise aisément un système à n macro-analyseurs fonctionnant en parallèle chacun à partir de sa propre macro-grammaire. Ce système démarre avec un certain macro-analyseur sur la chaîne d'entrée complète et les autres sur une chaîne vide. Lorsqu'une macro est appliquée

par le premier analyseur, des phrases peuvent être ajoutées à de nouvelles chaînes d'entrée pour les autres processeurs qui commencent alors leur(s) analyse(s). Le transfert de phrases se poursuit jusqu'à la fin de toutes les analyses. Le (ou les) langage(s) cible(s) peuvent être définis arbitrairement comme la (ou les) analyse(s) finale(s) d'un (ou de plusieurs) processeur(s).

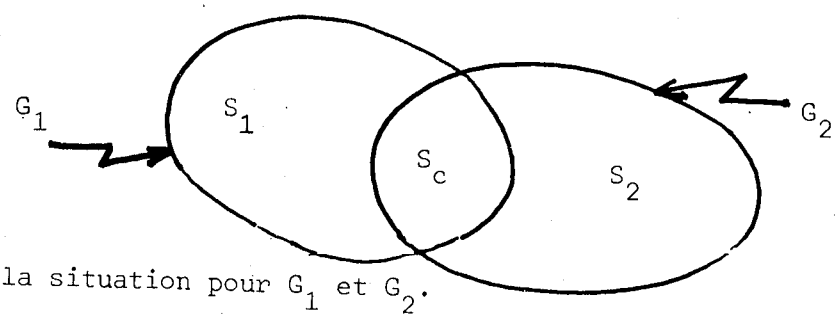
Une difficulté potentielle de cette solution est que les grammaires pour chacun des analyseurs ne sont pas indépendantes les unes des autres sauf (et seulement si) les phrases transférées sont composées seulement de terminaux. On conçoit que le transfert de non-terminaux est très utile mais que le mécanisme doit être défini avec précision.

La plus logique des définitions possibles semble être que deux non-terminaux pour deux grammaires différentes sont identiques s'ils définissent exactement la même chaîne terminale. Malheureusement ceci exige, de la part du méta-analyseur, de résoudre le problème de l'équivalence de deux langages à partir de deux grammaires (indécidable).

Une simplification possible, serait de définir l'égalité de deux non-terminaux par les composants de leurs productions (c'est une définition récursive) implémenté directement de cette manière, on aboutit encore à un problème très lourd de traitement de listes pour le méta-analyseur. On évite cela en notant spécifiquement que certaines règles appartiennent à deux (ou plusieurs) grammaires - ce qui restreint bien sûr le problème.

7.21 - La notation -

Soient deux grammaires G_1 et G_2 qui partagent certaines règles. Chacune (G_1 ou G_2) est constituée de règles propres (S_1 ou S_2) et de règles communes (S_c). Le diagramme suivant :



représente la situation pour G_1 et G_2.

Considérons maintenant les productions pour les non-terminaux de S_1 : elles peuvent inclure tout non-terminal défini en G_1, c'est-à-dire S_1 ou S_c. Une assertion analogue est vraie pour S_2. Pour S_c, les non-terminaux ne doivent inclure cependant que des non-terminaux de S_c (sinon S_c ne serait pas le sous-langage commun). En d'autres termes, les références peuvent être utilisées que dans une direction :

de S_1 ou S_2 vers S_c.

Ceci rappelle la définition des portées dans des langages à structure de bloc et la même dénotation est très utile ici. Le diagramme se réécrit à la manière d'un ensemble de grammaires :

```
début
    début
        <règles S1>
    fin ;
    début
        <règles S2>
    fin ;
    <règles Sc>
fin
```

Rappelons que les non-terminaux en partie gauche des règles sont des déclarations et que les parties droites sont composées de références à d'autres non-terminaux. Ainsi, la partie droite d'une règle en S_1 peut se composer de références à S_1 et S_c. Mais une règle S_c peut se référer seulement à son propre bloc, ce qui est l'effet désiré.

Cette formalisation est extrêmement commode pour le méta-analyseur puisque le seul contrôle nécessaire est la vérification de la portée de bloc qui existe déjà (paragraphe 1.4). De plus, la définition par défaut de l'axiome de chaque analyseur est simple : c'est la première règle qui le donne en partie gauche pour les blocs les plus internes. Le dernier problème de notation est la manière de définir quelle(s) partie(s) de remplacement donner à quel(s) analyseur(s). Chaque grammaire recevra un nom par une

étiquette introduisant le bloc begin pour ce faire. La portion de la phrase de remplacement à donner à un analyseur pour cette grammaire sera associée à ce nom.

Par exemple, une macro-règle appartenant à la grammaire G_1 s'écrira :

$$N \rightarrow U_1\ U_2\ U_3$$
$$= U_i\ U_j$$
$$[g_2 : U_r\ U_l\ U_m]$$
$$[g_3 : U_n\ U_o]$$

qui signifie : retenir la phrase "$U_i\ U_j$" pour l'analyseur de g_1 mais la phrase "$U_k\ U_l\ U_m$" sera passée à g_2 et "$U_n\ U_o$" à g_3.

Supposons g_1, g_2 et g_3 au même "niveau de bloc". On doit alors prendre soin de noter que les non-terminaux pour les phrases passées sont tous définis à quelque niveau plus haut (un non-terminal lui-même défini en g_1 n'est pas connu en g_2 ou g_3).

7.22 - Un exemple -

La traduction d'infixée vers BL/1 sera détaillée ici en utilisant la méthode à grammaires parallèles avec la notation décrite ci-dessus.

Deux grammaires seront utilisées. La première est la grammaire d'origine faite de macro-règles pour décrire les affectations PL/1. La seconde est simplement la grammaire BL/1 définie en appendice C sans aucune macro. Les macros de la grammaire PL/1 envoient des portions de leurs remplacements vers la chaîne d'entrée BL/1. Il n'y a pas de sous-grammaire commune à proprement parlé, si ce n'est les reconnaisseurs lexicographiques tels que <identificateur>, ...

La forme générale de description des grammaires est par conséquent :

```
  début
PL/1 : début
          <règles PL/1>
       fin ;
BL/1 : début
          listinstr → instr ';' | listinstr instr ';' ;
          instr → 'pass' descripteur 'to' descripteur |
                  'add' descripteur ';' descripteur '→' descripteur |
                  ... ;
          descripteur → identificateur | ...
       fin ;
commun : identificateur → * L n ;
         ...
  fin
```

On considère maintenant la notation des "règles PL/1". En commençant au niveau affectation on a :

```
instr PL/1 → idf_g '=' expr ';'
           = si expr →* idf_d
             alors [BL/1 : 'pass' idf_d 'to' idf_g]
             fsi
```

Cette macro vérifie simplement que l'affectation comporte un seul identificateur en partie droite du signe égal ('='). Si c'est le cas, une instruction simple est fournie à l'analyseur BL/1 : passer la valeur d'une variable <idf>$_g$ à l'autre <idf>$_d$. Le remplacement pour l'analyseur PL/1 est vide ce qui signifie que ce dernier va trouver une autre instruction PL/1 (<instr PL/1>) dans ce qui reste de la chaîne d'entrée PL/1.

Que se passe-t-il lorsque l'expression <expr> n'est pas un identificateur simple ? Il est assez surprenant de constater que la situation ne peut se présenter puisque toute expression complexe est déjà réduite par des appels antécédents des macros suivantes : (les noms utilisés sont ceux du paragraphe 3.21) :

$expr \rightarrow expr_g$ '|' $opd2_d$

= si ($expr_g \overset{*}{\rightarrow} idf_g$) $\wedge$ ($opd2_d \overset{*}{\rightarrow} idf_d$)

alorsi idf_t : NOUVEAU

alors idf_t

[BL/1 : 'or' idf_g ',' idf_d '→' idf_t ';']

fsi

L'opération PL/1 "ou" (|) a la priorité la plus haute c'est donc la première macro écrite. Les prédicats vérifient que les deux parties du signe ou sont des identificateurs simples ; si c'est le cas, le nouvel identificateur $<idf>_t$ est créé.

La phrase originale :

$<idf>_g$ | $<idf>_d$

est remplacée par $<idf>_t$ et une instruction BL/1 or est envoyée à l'analyseur BL/1. A noter que ceci accomplit les propos de départ à savoir : la suppression des instructions BL/1 de l'expression infixée modifiée.

Une règle de transition (ou macro sans clause remplacement) doit être ajoutée également à la grammaire PL/1 :

$expr \rightarrow opd2$

Des paires similaires de macros doivent être écrites pour chaque niveau de précédence dans la grammaire infixée jusqu'au primaire <opd8> :

$opd8 \rightarrow idf$

Les primaires parenthésés peuvent être manipulés en otant simplement les parenthèses une fois leur contenu réduit à un identificateur simple :

```
opd8 → '(' expr ')'
     = si expr →* idf
       alors idf
       fsi
```

7.3 - Commentaires -

La proposition ci-dessus pour des grammaires parallèles n'est qu'un projet. Si elle apparaît utile sur le papier, elle n'a pas encore été essayée sur machine. Un nombre très restreint d'essais ont été considérés ; aussi aucune estimation réelle de leur utilité potentielle n'a été faite. Il y a tout lieu de croire qu'on pourrait l'implémenter aisément en modifiant le macro-processeur existant. L'algorithme d'Earley [9] sur lequel repose notre implémentation se prête en lui-même à des modifications ce qui est le coeur du problème. Le reste de la question consiste à prendre en considération la notation nécessaire ; c'est assez aisé compte-tenu des méthodes décrites en 1.52 et qu'on peut réutiliser.

Le "parallélisme" a été évoqué souvent dans ce chapitre et mérite quelque attention. Si on traite les analyseurs comme des processeurs asynchrones alors il faut nécessairement un mécanisme de synchronisation. Les sémaphores - du type de ceux introduits par Dijkstra - peuvent certainement être introduits soit comme méta-objets pour la chaîne d'entrée soit sous forme de fonctions-Prédicat.

En fait tout le champ d'application du traitement collatéral devient applicable et on doit se demander si toutes ces techniques sont applicables pour les problèmes à résoudre. Jusqu'ici les exemples traités sont toujours des systèmes à deux grammaires avec presque toujours une seule direction de transfert. Une étude plus approfondie, dans un premier temps, serait celle d'ajouter à une grammaire un mécanisme d'empilement ("push-down"), c'est-à-dire un espace temporaire ; là encore la synchronisation peut ne pas être explicite.

Le problème de la généralisation du schéma de "grammaires parallèles" ne peut être raisonnablement abordé sans une étude plus approfondie. La direction choisie ici est uniquement une indication du travail à effecteur encore sur ce type de macro-processeur.

8 - CONCLUSION -

D'une certaine manière, le contenu de cette thèse semble rester sur un paradoxe apparent : bien que le principal sujet soit la sémantique des langages de calculateurs, l'outil le plus fréquemment utilisé s'appelle une "macro-syntaxique". Ce sont les informaticiens qui en sont responsables car ils ont tendances à considérer la "syntaxe" comme étant les structures pouvant être définies par des règles BNF plutôt que "la partie de la signification d'un texte qui est exprimée par l'ordre de juxtaposition et la composition des constituants de ce texte", ceci étant la définition linguistique la plus communément acceptée. La différence clé réside dans les mots forme et signification. L'informaticien range généralement tout ce qui concerne la signification (ou la traduction, ce qui est équivalent) sous le vocable "sémantique".

Les macros-syntaxiques font ressortir cette distinction. En effet, lorsqu'un texte PL/1 est analysé comme étant un texte BL/1 au moyen des macros-syntaxiques, une analyse sémantique partielle (au sens linguistique du terme) est réalisée. Cette analyse est nécessairement partielle, car il y a toujours un reliquat de texte objet qui possède de la sémantique non résolue. Ce problème doit sembler plus évident à l'auteur d'un dictionnaire monolingue qui doit définir les mots d'un langage avec un sous-ensemble de ce même langage (c'est-à-dire, écrire des macros), qu'à un écrivain de compilateur qui a tendance à regarder "la machine" et son langage comme données.

Un certain nombre d'exemples, de la technique de définition - par - traduction, ont été donnés dans cette thèse : langage de base PL/1, métalangage, modifications ALGOL 68. Ces exemples ont en commun la notion que le texte source a été écrit sous une forme abrégée (particulièrement pratique) du langage objet et que l'analyseur de macro est capable de découvrir ces abréviations lorsqu'il le désire.

Tout implémenteur sait qu'une bonne définition n'est malheureusement qu'une partie du problème en face duquel il se trouve. C'est pourquoi, dans la mesure du possible, j'ai essayé d'évaluer et de tester les exemples présentés dans cette thèse. Ceci a été fait assez à fond, par exemple, pour le métalangage (primitivement grâce au travail de Messieurs Ph. CHATELIN et P. COUSOT) avec des résultats très encourageants : le métalangage s'est révélé très souple et a produit, en même temps, du "code" (dans le cas présent : des grammaires) qui étaient aussi efficaces que si elles avaient été produites par des moyens classiques.

BL/1 a aussi été partiellement implémenté par le travail précieux de Monsieur D. SERAIN. Les résultats sont, ici, beaucoup plus difficiles à juger car une grande partie du problème était de trouver une définition appropriée pour le langage de base PL/1. Ce qui a été implémenté, initialement, a montré, à la fois, les forces et les faiblesses de l'approche.

Quelques macros pour la traduction de PL/1 en BL/1 ont été écrites beaucoup plus récemment et sont à considérer pour un nouvel essai de définition.

Toute discussion des problèmes d'implémentation chevauche, peut être inévitablement, ceux de code intermédiaire (que je préfère appeler langage intermédiaire). Un des avantages cachés de l'utilisation des macros syntaxiques est qu'elle concentre l'attention de l'utilisateur sur les propriétés linguistiques du texte produit ; ce n'est pas simplement du "code". Inversement, on peut aller trop loin dans cette voie. Une des difficultés, que nous avons rencontrée en implémentant ALGOL 68, était d'imaginer une méthode n'obligeant le texte à être, tout entier, en mémoire principale. Le schéma de segmentation de texte présenté n'est manifestement pas linguistique si on regarde le langage intermédiaire. Néanmoins, il est basé sur la connaissance d'une caractéristique du langage, c'est-à-dire, sur le fait que les unités neutres peuvent être retirées du texte sans aucun effet sur l'environnement auquel elles appartiennent.

La traduction syntaxique ne résout qu'une partie de la sémantique et laisse donc un résidu devant être pris en charge par d'autres méthodes. C'est pourquoi, j'ai aussi parlé, dans cette thèse, de quelques problèmes entraînant une génération de code supplémentaire. En particulier, un schéma des "attributs de position" a été développé dans lequel une structure de contrôle de type arborescente a été utilisée pour la détermination des optimisations locales des instructions machine. Bien que je ne l'aie pas fait souligner dans le chapitre 6, la méthode présente une ressemblance logique frappante avec la méthode des traductions, servant à déterminer les modifications, dans le chapitre 5. Cela signifie qu'elles utilisent toutes deux des indicateurs attachés aux noeuds de l'arbre, l'une pour le mode, l'autre pour les attributs de position et que la connaissance des noeuds voisins sert à déterminer la valeur de l'indicateur pour le noeud courant.

Néanmoins, j'ai traité un cas (celui des modifications) comme s'il s'agissait d'une "traduction" et l'autre cas (celui des positions) comme s'il s'agissait d'un "résidu". Cela ne conduit-il pas au caractère artificiel de cette distinction ? En fait, comme j'essaye de le montrer dans la partie sur les "niveaux de langages" (2.00) tout langage cible peut devenir le langage source pour un nouveau niveau de traduction. La manière de prendre en charge certains travaux sémantiques a toujours été guidé par d'autres considérations : clarté, facilité d'implémentation, caractère raisonnable ... etc ....

Seul le chapitre 4 (sur la segmentation de l'arbre) est peut être un exemple où les techniques de traduction ne sont pas applicables, car ce qui est traité est, en réalité, la structure de données du programme compilant et pas du tout le processus de traduction.

J'espère qu'il est clair, que les chapitres 4, 5 et 6, tout en étant un support pour la présentation de certaines idées sémantiques, représentent aussi quelques-uns des résultats obtenus par notre travail sur la réalisation d'un compilateur ALGOL 68 à Grenoble. De ce point de vue, les macros et les transformations syntaxiques servent simplement de méthode documentaire.

Nos tentatives d'utilisation des macros dans des implémentations réelles nous a montré clairement où des améliorations étaient possibles. Pour implémenter les processus de traduction proposés, il est nécessaire d'avoir une méthode d'utilisation de macros très puissante. Il semble que les macros-syntaxiques, proposées par Monsieur S. SCHUMAN, soient les plus appropriées dans la majorité des cas. Cependant, il y a des traductions difficiles à réaliser avec les macros-syntaxiques et, en conséquence, des modifications dans la définition des macros sont à prévoir - la méthode des grammaires en parallèle dans le chapitre 7 étant un de ces points.

Un point qui semble évident mais qui est mal mentionné est le suivant : étant donné que les primitives du langage de base sont indépendantes de la machine, les résultats de notre travail par macros est universellement applicable. Je pense qu'il est exact de dire que ce souci est commun à tous les membres grenoblois travaillant sur les langages extensibles et les macros.

Les thèses de Monsieur D. BERT [2], de Madame V. BAJAR [1] et de Monsieur P. COUSOT [5] l'ont déjà montrées et je suis sûr que les suivantes continueront à confirmer cette tendance.

A - EXEMPLES DE MACROS-SYNTAXIQUES -

A.0 - Notation -

Les exemples montrés ci-dessous ont tous été testés sur notre système. Les résultats sont tels qu'ils sortent du terminal, c'est-à-dire avec certaines abréviations pour faciliter la frappe.

Les principaux sont :

si	(	sinon	|
alors	|	sinsi	|:
alorsi	|:	fsi	)

Donc la phrase de remplacement :

si P alors R sinsi P2 alors R2 sinon R3 fsi

s'écrit :

(P | R |: P2 | R2 | R3).

La notation utilisée pour les macros-règles est définie formellement par la métagrammaire donnée en appendice B.

En général, il y a quatre parties pour chaque exemple :

(1) - la macro, sous forme d'un fichier d'entrée CMS,

(2) - une chaîne d'entrée acceptée par le macro-analyseur,

(3) - un arbre syntaxique représentant l'analyse en grammaire de base de cette chaîne,

(4) - une chaîne de sortie correspondant à l'arbre.

A noter que l'arbre syntaxique est représenté la racine à gauche et chaque descente d'un nouveau niveau occupant une nouvelle colonne. Donc la représentation est de gauche à droite au lieu d'être de haut en bas.

Les appels de fonction-prédicat "nouveau" fournissent de nouveaux identificateurs. Puisque ceux-ci sont sensés être uniques, une représentation spéciale est utilisée dans les sorties de l'analyseur. Le premier caractère de chaque nouvel identificateur est "sous-ligné" (_) suivi par un nombre. Ainsi le treizième identificateur produit par la fonction-prédicat est :

_13

A.1 - Infixé vers préfixé -

C'est une version réduite de la macro discutée en section 3.2122.

Abréviations :

SL - liste d'instructions
ST - instruction
OD - opérande
ID - identificateur
UO - opérateur unaire
BO - opérateur binaire
EX - expression
SP - primaire signé
PR - primaire

A.11 - La macrogrammaire -

```
SL -> SL ST ';' | ST ';' ;
ST -> OD;
OD -> ID |UO OD | BO OD OD ;
UO -> '+' | '-' ;
BO -> '+' | '*' | '=' ;
ID -> *L1;
ST -> ID '=' EX ';'
   =  '=' ID '!T!' EX ';' ;
EX -> EX BO SP | SP ;
OD -> '!T!' EX
   =  (  EX -> EX.2 BO SP.2
      |  BO '!T!' EX.2 '!T2!' SP.2
      |: EX -> SP.2
      |  '!T2!' SP.2
      );
SP -> UO SP | PR ;
OD -> '!T2!' SP
   =  (  SP -> UO SP.2
      |  UO '!T2!' SP.2
      |: SP -> PR.2
      |  '!T3!' PR.2
      );
PR -> '(' EX ')' | ID ;
OD -> '!T3!' PR
   =  ( PR -> ID
      |  ID
      |: PR -> '(' EX ')'
      | '!T!' EX
      );
```

A.12 - Une chaîne d'entrée -

```
a = b * (c + d) * e;
```

A.13 - L'arbre d'analyse de base -

```
SL ST OD BO =
        OD =ID A
        OD BO *
           OD BO *
              OD =ID B
              OD BO +
                 OD =ID C
                 OD =ID D
           OD =ID E
   ;
```

A.14 - La chaîne de sortie -

```
= A ** B + C  D  E ;
```

A.2 - Préfixé vers BL/1 -

C'est une version réduite de la macro discutée en section 3.2121.

Abréviations :

SL - liste d'instructions
ST - instruction
ID - identificateur
OD - opérande
UO - opérateur unaire
BO - opérateur binaire
NEW- nouvel identificateur

A.21 - La macrogrammaire -

```
SL -> SL ST ';' | ST ';' ;
ST -> '!ADD!' ID ',' ID '->' ID | '!MULT!' ID ',' ID '->' ID |
   '!PASS!' ID '->' ID | '!MINUS!' ID '->' ID ;
ID ->  *L1;
OD -> ID | UO OD | BO OD OD;
UO -> '+' | '-' ;
BO -> '+' | '*' ;
ST -> '=' ID OD ';'
   =  (  OD -> BO OD.2 OD.3
      |  (  OD.2 -> ID.2
         |  (  OD.3 -> ID.3
            |  (  BO -> '+'
               |  '!ADD!' ID.2 ',' ID.3 '->' ID ';'
               |: BO -> '*'
               |  '!MULT!' ID.2 ',' ID.3 '->' ID ';'
               )
            |: ID.4 : NEW
            |  '=' ID.4 OD.3 ';'
               '=' ID BO ID.2 ID.4 ';'
            )
         |: ID.4 : NEW
         |  '=' ID.4 OD.2 ';'
            '=' ID BO ID.4 OD.3 ';'
         )
      |: OD -> UO OD.2
      |  (  UO -> '+'
         |  '=' ID OD.2 ';'
         |: UO -> '-'
         |  (  ID.4 : NEW
            |  '=' ID.4 OD.2 ';'
               '!MINUS!' ID.4 '->' ID ';'
            )
         )
      |: OD -> ID.2
      |  '!PASS!' ID.2 '->' ID ';'
      );
NEW -> *F2;
```

A.22 - Une chaîne d'entrée -

```
= a * * b + c d e ;
```

A.23 - L'arbre d'analyse de base -

```
SL SL SL ST !ADD!
            =ID C
            ,
            =ID D
            ->
            =ID _4
         ;
      ST !MULT!
         =ID B
         ,
         =ID _4
         ->
         =ID _3
      ;
   ST !MULT!
      =ID _3
      ,
      =ID E
      ->
      =ID A
   ;
```

A.24 - La chaîne de sortie -

```
!ADD! C , D  -> _4  ;
!MULT! B , _4  -> _3  ;
!MULT! _3  , E  -> A ;
```

A.3 - La combinaison des deux exemples précédents -

Par addition des macrogrammaires en sections A.1 et A.2 on obtient une macrogrammaire capable de traduire le code infixé directement en langage de base.

A.31 - La macrogrammaire -

```
SL -> SL ST ';' | ST ';' ;
ST -> '!ADD!' ID ',' ID '->' ID | '!MULT!' ID ',' ID '->' ID |
   '!PASS!' ID '->' ID | '!MINUS!' ID '->' ID ;
ID ->  *L1;
OD -> ID | UO OD | BO OD OD;
UO -> '+' | '-' ;
BO -> '+' | '*' | '=' ;
ST -> '=' ID OD ';'
   =  (  OD -> BO OD.2 OD.3
      |  (  OD.2 -> ID.2
         |  (  OD.3 -> ID.3
            |  (  BO -> '+'
               |  '!ADD!' ID.2 ',' ID.3 '->' ID ';'
               |: BO -> '*'
               |  '!MULT!' ID.2 ',' ID.3 '->' ID ';'
               )
            |: ID.4 : NEW
            |  '=' ID.4 OD.3 ';'
               '=' ID BO ID.2 ID.4 ';'
            )
         |: ID.4 : NEW
         |  '=' ID.4 OD.2 ';'
            '=' ID BO ID.4 OD.3 ';'
         )
      |: OD -> UO OD.2
      |  (  UO -> '+'
         |  '=' ID OD.2 ';'
         |: UO -> '-'
         |  (  ID.4 : NEW
            |  '=' ID.4 OD.2 ';'
               '!MINUS!' ID.4 '->' ID ';'
            )
         )
      |: OD -> ID.2
      |  '!PASS!' ID.2 '->' ID ';'
      );
NEW -> *F2;
ST -> ID '=' EX ';'
   =  '=' ID '!T!' EX ';' ;
EX -> EX BO SP | SP ;
OD -> '!T!' EX
   =  (  EX -> EX.2 BO SP.2
      |  BO '!T!' EX.2 '!T2!' SP.2
      |: EX -> SP.2
      |  '!T2!' SP.2
      );
SP -> UO SP | PR ;
OD -> '!T2!' SP
   =  (  SP -> UO SP.2
      |  UO '!T2!' SP.2
      |: SP -> PR.2
      |  '!T3!' PR.2
      );
```

```
PR -> '(' EX ')' | ID ;
OD -> '!T3!' PR
   =  (  PR -> ID
      |  ID
      |: PR -> '(' EX ')'
      |  '!T!' EX
      );
```

A.32 - Une chaîne d'entrée -

```
a = b * (c + d) * e ;
```

A.33 - L'arbre d'analyse de base -

```
SL SL SL ST !ADD!
            =ID C
            ,
            =ID D
            ->
            =ID _6
         ;
      ST !MULT!
         =ID B
         ,
         =ID _6
         ->
         =ID _5
      ;
   ST !MULT!
      =ID _5
      ,
      =ID E
      ->
      =ID A
   ;
```

A.34 - La chaîne de sortie -

```
!ADD! C , D  -> _6  ;
!MULT! B , _6  -> _5  ;
!MULT! _5  , E  -> A ;
```

A.4 - "Cross-sections" -

C'est la réalisation des macros discutées en section 3.22.

Abréviations :

SL - liste d'instructions

STAT - instruction

DES - destination d'une affectation

EX - expression

ID - identificateur

IL - liste d'indices

IX - indice

AO - opérateur additatif

T - terme

UO - opérateur unaire

P - primaire

NU - nombre

SCAN - scruter

INCR - incrémenter

NEWID - nouvel identificateur

FOUND - trouvé

NOT FOUND - pas trouvé

REPLACE - remplacer

ENDR - fin remplacement

A.41 - La macrogrammaire -

```
SL -> SL STAT ';' | STAT ';' ;
STAT -> '!2!' DES '=' EX | 'DO ' ID '=' EX 'TO' EX.2 | 'END';
DES -> ID | ID '(' IL ')';
IL -> IX ',' IL | IX;
IX -> EX | '*';
EX -> EX AO T | T;
AO -> '+' | '-';
T -> UO T | P;
UO -> '+' | '-';
P -> '(' EX ')' | ID '(' IL ')' | ID | NU;
ID -> *L1;
NU -> *L2;
STAT -> DES '=' EX ';'
   = ( DES -> ID '(' IL ')'
     | ID '( !SCAN! )' IL ')=' EX ';'
     |: DES -> ID
     | '!2!' DES '=' EX ';'
     );
IL -> '!SCAN!' NU IL.2
   = ( INCR(NU)
     |: IL.2 -> IX ',' IL.3
     | ( IX -> '*'
       | ( ID : NEWID | ID ',' IL.3 '!FOUND!' NU ID )
       | IX ',' '!SCAN!' NU IL.3)
     |: IL.2 -> IX
     | ( IX -> '*'
       | ( ID : NEWID | ID '!FOUND!' NU ID )
       | IL.2 '!NOTFOUND!'
       )
     );
STAT -> ID '(' IL '!NOTFOUND!' ')=' EX = '!2!' ID '(' IL ')=' EX;
STAT -> ID '(' IL '!FOUND!' NU ID.2 ')=' EX
   = 'DO ' ID.2 '=LBOUND(' ID ',' NU ')TO HBOUND(' ID ',' NU ');'
     ID '(' IL ')=' '!REPLACE!' ID.2 EX '!ENDR!' ';'
     'END';
INCR->*F1; NEWID->*F2;
EX -> '!REPLACE!' ID EX.2 '!ENDR!'
   = (EX.2 -> EX.3 AO T | '!REPLACE!' ID EX.3 '!ENDR!' AO '!R2!' ID T
   |: EX.2 -> T | '!R2!' ID T);
T -> '!R2!' ID T.2
   = (T.2 -> UO T.3 | UO '!R2!' ID T.3 |: T.2 -> P | '!R3!' ID P);
P -> '!R3!' ID P.2
   = (P.2 -> '(' EX ')' | '(' '!REPLACE!' ID EX '!ENDR!' ')'
   |: P.2 -> ID.2 '(' IL ')' | ID.2 '(' '!RIX!' ID IL ')'
   |: P.2 -> ID.2 | ID.2
   |: P.2 -> NU | NU);
IL -> '!RIX!' ID IL.2
   = ( IL.2 -> IX ',' IL.3
     | ( IX -> '*' | ID ',' IL.3 | IX ',' '!RIX!' ID IL.3 )
     |: IL.2 -> IX |: IX -> '*' | ID | IX );
```

A.42 - Une chaîne d'entrée -

```
a (*, 1, *) = - b (*) + c (d, e, *, *) ;
```

A.43 - L'arbre d'analyse de base -

```
*
  SL SL SL SL SL STAT DO
                      =ID _1
                      =
                      EX T P =ID LBOUND
                             (
                             IL IX EX T P =ID A
                                ,
                                IL IX EX T P =NU 1
                             )
                      TO
                      EX T P =ID HBOUND
                             (
                             IL IX EX T P =ID A
                                ,
                                IL IX EX T P =NU 1
                             )
                  ;
             STAT DO
                  =ID _2
                  =
                  EX T P =ID LBOUND
                         (
                         IL IX EX T P =ID A
                            ,
                            IL IX EX T P =NU 3
                         )
                  TO
                  EX T P =ID HBOUND
                         (
                         IL IX EX T P =ID A
                            ,
                            IL IX EX T P =NU 3
                         )
             ;
```

(Suite de l'arbre d'analyse de base)

```
                    STAT !2!
                        DES =ID A
                            (
                            IL IX EX T P =ID _1
                                ,
                                IL IX EX T P =NU 1
                                    ,
                                    IL IX EX T P =ID _2
                            )
                        =
                        EX EX T UO -
                                T P =ID B
                                    (
                                    IL IX EX T P =ID _1
                                    )
                           AO +
                           T P =ID C
                               (
                               IL IX EX T P =ID D
                                   ,
                                   IL IX EX T P =ID E
                                       ,
                                       IL IX EX T P =ID _1
                                           ,
                                           IL IX EX T P =ID _2
                               )
                    ;
                STAT END
            ;
        STAT END
        ;
```

A.44 - La chaîne de sortie -

```
DO  _1  = LBOUND ( A , 1 ) TO HBOUND ( A , 1 );
DO  _2  = LBOUND ( A , 3 ) TO HBOUND ( A , 3 );
!2! A ( _1  , 1 , _2  )=- B ( _1  )+ C ( D , E , _1  , _2  );
END;
END;
```

A.45 - Une autre chaîne d'entrée -

```
a (*, *, X + 1, *) = b (X - 1, *, *,) - c (u, *, *, *) ;
```

A.46 - La chaîne de sortie -

```
DO _3 = LBOUND ( A , 1 ) TO HBOUND ( A , 1 );
DO _4 = LBOUND ( A , 2 ) TO HBOUND ( A , 2 );
DO _5 = LBOUND ( A , 4 ) TO HBOUND ( A , 4 );
12! A ( _3 , _4 , X + 1 , _5 )= B ( X - 1 , _3 , _4 )- C ( U ,
_3 , _4 , _5 );
)END;
END;
END;
```

B - META-GRAMMAIRES -

Comme indiqué dans la section 1.5 le système de préparation des macrogrammaires est implanté au moyen de macrogrammaires elles-mêmes. La génération téléscopique commence avec une méta-grammaire capable de se décrire elle-même et par étapes successives arrive à une version beaucoup plus puissante. Cet appendice explicitera ce processus.

B.1 - La méta-grammaire de base -

Abréviations :

G - grammaire

L - liste de règles

R - règle

PR - règle propre

NU - nombre

NA - nom d'identificateur

P - phrase

U - unité syntaxique

STR - chaîne de caractères

B.11 - La grammaire de base comme chaîne d'entrée -

```
G  -> L ';' *I 1 ;
L  -> R ;
L  -> L ';' R ;
R  -> PR ;
R  -> PR '*I' NU  *I 7 ;
R  -> NA '->''*L' NU  *I 13 ;
PR  -> NA '->' P  *I 8 ;
P  -> U ;
P  -> P  U ;
U  -> NA  *I 11 ;
U  -> STR  *I 12 ;
NA  -> *L 1 ;
NU  -> *L 2 ;
STR  -> *L 3 ;
```

B.12 - L'arbre de la grammaire de base (analysée par elle-même) -

```
G L L L L L L L L L L L L L L R PR =NA G
                                  ->
                                    P P U =NA L
                                      U =STR ';'
                                *|
                                =NU 1
                             ;
                             R PR =NA L
                                  ->
                                    P U =NA R
                          ;
                          R PR =NA L
                               ->
                                 P P P U =NA L
                                     U =STR ';'
                                   U =NA R
                       ;
                       R PR =NA R
                            ->
                              P U =NA PR
                    ;
                    R PR =NA R
                         ->
                           P P P U =NA PR
                               U =STR '*|'
                             U =NA NU
                       *|
                       =NU 7
                 ;
                 R PR =NA R
                      ->
                        P P P P U =NA NA
                            U =STR '->'
                          U =STR '*L'
                        U =NA NU
                   *|
                   =NU 13
              ;
              R PR =NA PR
                   ->
                     P P P U =NA NA
                         U =STR '->'
                       U =NA P
                 *|
                 =NU 3
           ;
           R PR =NA P
                ->
                  P U =NA U
     ;
     R PR =NA P
          ->
```

(Suite de l'arbre de la grammaire de base)

```
                    U =NA U
                ;
                R PR =NA U
                     ->
                     P U =NA NA
                  *I
                  =NU 11
              ;
              R PR =NA U
                   ->
                   P U =NA STR
                *I
                =NU 12
            ;
            R =NA NA
              ->
              *L
              =NU 1
          ;
          R =NA NU
            ->
            *L
            =NU 2
      ;
      R =NA STR
        ->
        *L
        =NU 3
;
```

B.13 - Les numéros de synthèse -

Pour la méta-grammaire de base les fonctions de synthèse sont exploitées pour produire des grammaires, en codage interne (numérique). Voici les numéros utilisés et leur signification :

(numéro)	(génération)
0 (par défaut)	appel de tous les sous-non-terminaux de gauche à droite
1	initialisation du fichier destiné à la grammaire codée, deux appels de <liste de règles>, terminaison du fichier
7	appel de<règle propre>, codage de numéro de synthèse explicite
8	premier passage : déclaration de non-terminal à gauche comme occurrence de définition deuxième passage : appel de <phrase>, terminaison de codage de règle propre
11	codage de non-terminal de phrase
12	codage de terminal de phrase
13	premier passage : déclaration de reconnaisseur lexicographique, codage du même deuxième passage : néant

B.2 - Première amélioration au méta-langage -

Le méta-langage de base ne permet que l'écriture de règles hors-contexte simples. La première modification est donc de permettre des macros-règles. Notons que la grammaire ci-dessous peut être analysée et synthétisée par la grammaire de base.

B.21 - La nouvelle grammaire comme chaîne d'entrée -

Abréviations (nouvelles) :

SR - remplacement simple

RP - phrase de remplacement

RU - unité syntaxique de remplacement

```
G -> L ';' *I1;
L -> L ';' R;
L -> R;
R -> PR;
R -> PR '*I' NU *I7;
R -> NA '->' '*L' NU *I13;
R -> PR '=' SR *I5;
PR -> NA '->' P *I8;
P -> P U;
P -> U;
U -> NA *I11;
U -> NA '.' NU *I4;
U -> STR *I12;
SR -> RP *I14;
RP -> RP RU;
RP -> RU;
RU -> NA *I11;
RU -> NA '.' NU *I4;
RU -> STR *I12;
NA -> *L1;
NU -> *L2;
STR -> *L3;
```

B.22 - Additions aux numéros de synthèse -

(numéro)	(génération)
4	codage de non-terminal indexé
5	premier passage : appel de <règle propre>
	deuxième passage : appel de <règle propre>, appel de <remplacement simple>, terminaison de codage de macro-règle
14	appel de <phrase de remplacement>, terminaison de codage de phrase

B.3 - Deuxième amélioration au méta-langage -

Une fois le méta-langage étendu pour permettre l'écriture de macrorègles, il devient possible d'ajouter des extensions purement syntaxiques par voie de méta-macros. C'est ce qui est fait dans la méta-grammaire ci-dessous analysable par la grammaire précédente. Des macros sont ajoutées qui permettent d'analyser des règles et des macrorègles écrites avec la notation barre (|), cf. section 1.53. De plus, la phrase syntaxique vide est implantée en utilisant la notation : (). On remarquera qu'aucun nouveau numéro de synthèse n'est utilisé : donc le synthétiseur pour les deuxième et troisième méta-grammaires est formellement identique, bien que l'un est plus puissant que l'autre.

B.31 - La nouvelle grammaire comme chaîne d'entrée -

Abréviation (nouvelle) :

RHS - partie droite d'un règle hors-contexte.

```
G -> L ';' *I1;
L -> L ';' R;
L -> R;
R -> PR;
R -> PR '*I' NU *I7;
R -> NA '->' '*L' NU *I13;
R -> PR '=' SR *I5;
R -> NA '->' RHS '|' = NA '->' RHS ';' NA '->';
R -> NA '->' RHS '*I' NU '|' = NA '->' RHS '*I' NU ';' NA '->';
R -> NA '->' RHS '=' SR '|' = NA '->' RHS '=' SR ';' NA '->';
PR -> NA '->' RHS *I3;
RHS -> P;
RHS -> '()';
P -> P U;
P -> U;
U -> NA *I11;
U -> NA '.' NU *I4;
U -> STR *I12;
SR -> RP *I14;
RP -> RP RU;
RP -> RU;
RU -> NA *I11;
RU -> NA '.' NU *I4;
RU -> STR *I12;
NA -> *L1;
NU -> *L2;
STR -> *L3;
```

B.4 - Troisième amélioration au méta-langage -

Le deuxième méta-langage fournit un moyen pratique d'écrire la troisième méta-grammaire, qui est considérablement plus grande, parce que maintenant toutes les facilités manquantes vont être ajoutées : les remplacements conditionnels, l'échec, le sans, les abréviations de mot-clés pour les phrases conditionnelles, les expressions booléennes, la forme des fonctions-prédicats, les prédicats de sous-structure.

En effet, cette troisième étape constitue le dernier pas, parce que maintenant le mécanisme est disponible pour écrire n'importe quelle méta-grammaire.

B.41 - La nouvelle grammaire comme chaîne d'entrée -

Abréviations (nouvelles) :

RPL - remplacement

BE - expression booléenne

BT - terme booléen

BF - facteur booléen

BP - primaire booléen,

RES - résultat de fonction prédicat

ALT - alternative d'un remplacement

NT - non-terminal

ARO - flèche

APK - paquet d'arguments

AL - liste d'arguments

```
G -> L ';' *I1;
L -> R | L ';' R;
R -> PR *I9 | PR '*I' NU *I7 | PR '=' RPL *I5 |
     NA '->' '*L' NU *I13 | NA '->' '*F' NU *I2 |
     NA '->' RHS '|' = NA '->' RHS ';' NA '->' |
     NA '->' RHS '*I' NU '|' = NA '->' RHS '*I' NU ';' NA '->' |
     NA '->' RHS '=' RPL '|' = NA '->' RHS '=' RPL ';' NA '->';
PR -> NA '->' RHS *I8;
RHS -> P | '()' | ();
P -> U | P U;
U -> NA *I11 | NA '.' NU *I4 | STR *I12;
NA -> *L1; NU -> *L2; STR -> *L3;
RPL -> '(' BE ALT.1 ALT.2 ')' *I16 | SR *I14 | '¬' *I15 | '=' *I6 |
       '(' BE ALT ')' = '(' BE ALT '|¬)';
ALT -> '|' RPL | '|:' BE ALT ')' = '|(' BE ALT '))' |
       '|:' BE ALT.1 ALT.2 ')' = '|(' BE ALT.1 ALT.2 '))';
SR -> RP | '()' | ();
RP -> RU | RP RU;
RU -> NA *I11 | NA '.' NU *I4 | STR *I12 | '$' RU *I10;
BE -> BT | BE '+' BT *I17;
BT -> BF | BT '*' BF *I18;
BF -> BP | '¬' BP *I19;
BP -> '(' BE ')' *I20 | NT ARO RHS *I21 | RES NA APK *I3;
NT -> NA *I11 | NA '.' NU *I4;
RES -> NT ':' | ();
APK -> '(' AL ')' | ();
AL -> NT | AL ',' NT;
ARO  -> '->' *I22 | '=>' *I23;
```

B.42 - Additions aux numéros de synthèse -

(numéro)	(génération)
2	premier passage : déclaration de nom fonction-prédicat, codage de cette dernière. deuxième passage : néant
3	vérification du nom de fonction-prédicat, appel de <résultat>, appel de <paquet d'arguments>, terminaison du codage de fonction-prédicat.
6	codage du méta-symbole "sans" (=)
10	appel de <unité de remplacement>, codage du méta-symbole "effeuiller" qui décompose le non-terminal en une suite de terminaux
15	codage du méta-symbole "échec" ($\neg$)
16	appel de <expression booléenne>, appel de $<\text{alternative}>_1$, appel de $<\text{alternative}>_2$, terminaison du codage du remplacement conditionnel.
17	appel de <expression booléenne>, appel de <terme booléen>, terminaison du codage de l'opérateur "ou" (+)
18	appel de <terme booléen>, appel de <facteur booléen>, terminaison du codage de l'opérateur "et" ($*$)
19	appel de <primaire booléen>, terminaison du codage de l'opérateur "non" ($\neg$)
20	appel de <expression booléenne>
21	appel de <flèche>, appel de <non-terminal>, appel de <partie droite>, terminaison de codage du prédicat de sous-structure
22	signaler "sous-structure d'un seul niveau" ($\rightarrow$)
23	signaler "sous-structure profonde" ($\Rightarrow$)

C - GRAMMAIRE DE BL/1 -

```
prog → listeinstr 'edge' ;

listeinstr → instr ';' listeinstr   instr ';' ;

instr → 'describe' descr 'à' champadr ',' champent ':'
                    champattr ',' champent |
        'sketch' descr ':' champattr ',' champent |
        'layout' descr ':' champent |
        'frame' |
        'link' descr 'to' champadr |
        'value' attributionvaleur |
        'constant' descr '=' attributionconstante ',' champent |
        'label' descr |
        'fasten' descr |
        'share' descr 'as' literalcar |
        opérateurbinaire descrref ',' descrref '→' descr |
        opérateurunaire descrref '→' descr |
        'goto' descréti |
        'goif' descrbit 'to' descreti |
        'pass' descrref '→' descr |
        'change' descrref '→' descr |
        'shape' descr 'to' opindic descr descr |
        'stack' descr 'in' descrespace |
        'unstack' descr 'from' descrespace |
        'heap' descr 'in' descrespace |
        'unheap' descr 'from' descrespace |
```

```
        'clump' descr 'in' descrespace |
        'unclump' descr 'from' descrespace ;
champadr → descradr | 'base' descr | 'adr' descr ;
champattr → specattr | 'attr' descr ;
champent → descrent | literaldecimal | 'off' descr | 'rep' descr ;
descr → /* une identificateur */ ;
descradr → descr ; /* avec attribut adr */
descrent → descr ; /* avec attribut xb 15+0 */
descrespace → descr ; /* avec attribut s */
descréti → descr ; /* avec attribut l */
descrbit → descr ; /* avec attribut b */
descrcar → descr ; /* avec attribut c */
descrnombre → descr ; /* avec attribut fb, fd, xb ou xd */
descrgroupe → descr ; /* avec attribut g */
attributionvaleur → descrcar '=' literalcar |
                    descrbit '=' literalbit |
                    descrnombre '=' literalnombre |
                    descrgroupe '=' descr ;
attributionconstante → literalcar ':' 'c' |
                       literalbit ':' 'b' |
                       '0' ':' 'l' |
                       literalnombre ':' attrnombre |
                       descrgroupe ':' 'g' ;
literalcar → /* représentation d'une chaîne de caractères */ ;
literalbit → /* représentation d'une chaîne de bits */ ;
literalnombre → /* représentation d'un nombre */ ;
literaldecimal → /* représentation d'un nombre décimal */ ;
```

descrref → descr | 'base' descr | 'off' descr | 'rep' descr |
'adr' descr | 'size' descr ;

opérateurbinaire → 'add' | 'subtract' | 'multiply' | 'divide' |
'modulus' | 'equals' | 'larger' | 'smaller' |
'and' | 'or' | 'X-round' ;

opérateurunaire → 'minus' | 'not' | 'ceiling' | 'floor' |
'absolute' | 'F-round' ;

opindic → 'add' | 'subtract' | 'multiply' | 'divide' | 'modulus' ;

D - SPECIFICATIONS POUR LA TRADUCTION PL/1 VERS BL/1 -

Les notes suivantes sont un sommaire du travail que j'ai effectué pendant la conception originelle de BL/1. Mon intention était de déterminer les primitives sémantiques nécessaires au langage de base en écrivant informellement les transformations pour chaque instruction PL/1. Maintenant que BL/1 a été conçu et partiellement implanté et que les macros PL/1 ont été écrites, la tentation est grande de réécrire à la fois le langage de base et les spécifications de traduction. Comme on verra ces spécifications ne sont pas tout à fait complètes (par exemple, il manque des fonctions "built-in") ni sans erreurs, bien que je crois qu'elles fournissent un bon support pour l'argument que BL/1 est sémantiquement équivalent à PL/1.

La notation utilisée est essentiellement celle de la section 1.7. Les caractères terminaux sont tous PL/1 et donc majuscules. Ces miniscules sont utilisées pour les variables créées (en général leurs noms sont sensés être mnémoniques). Les non-terminaux sont indiqués par des crochets angulaires (<...>).

D.1 - Structure de contrôle -

D.11 - Do -

```
L : DO      ... { DO    } ... END L ;
    BEGIN       { BEGIN }
    PROC        { PROC  }

⇒ L : DO      ... { DO    } ... END ; END L ;
    BEGIN         { BEGIN }
    PROC          { PROC  }
```

; DO ; <listeinstr> ; END ; ⇒ ; <listeinstr> ;

```
DO ... A(I, J) = ... ⇒ t1 = I*  enjambée1 + J* enjambée2 ;
                        describe t2 @ A, t1 : attr t1, 1 ;
                        DO ... t2 = ...
```

```
DO I = B TO C, J = 1 BY -1 WHILE (BV) ; <listeinstr> ; END
   ⇒  t1 = 1 ;
      I  = B ;
      t2 = C ;
      GOTO   l1 ;
boucle(1) : I = I + 1
      l1 : IF I <= t2 THEN GOTO corps ;
      t1 = 2 ;
      t3 = -1 ;
      GOTO l2 ;
boucle(2) : J = J + t3 ;
      l2 : IF (BV)¬= 0 THEN GOTO corps ;
           GOTO dehors ;
      corps : <listeinstr> ;
           GOTO boucle (T1) ;
      dehors :
```

D.12 - Begin et End -

```
BEGIN ;
    retourprolog = prochaine ;
    DCL 1 bloc13 BASED (basebloc2) (pour le 13ème bloc sur 2ème niveau
                                    statique)
      2 anc num invoc FIXED,       (numéro d'invocation dynamique)
      2 anc epi LABEL,             (position de l'ancien épilogue)
      2 anc bd PTR,                (ancienne base de display)
      2 espace display,            (display)
          3 adr niveau 1 PTR,
               ...
          3 adr niveau n PTR,      (n = niveau statique courant = 2)
      2 <toutes déclarations AUTO et temporaire> ;
```

```
prologue : stack bloc13 in sysarea ;
           espacedisplay = display, BY NAME ;
           espacedisplay . adrniveau2 = basebloc ;
           anc b d = base display ;
           base display = ADR (espacedisplay) ;
           link bloc13 to display adrniveau2 ;
           anc num invoc = num invoc courant ;
           compte invoc = compte invoc + 1 ;
           num invoc courant = compte invoc ;
           anc epi = epicourant ;
           epicourant = epilogue ;
           <instructions supplémentaires de prologue>
           GOTO retour prologue ;

prochaine :

     END ; /* de bloc BEGIN */ =>
           retour épilogue = dehors ;
         épilogue : <instruction supplémentaires de épilogue>
           epicourant = anc epi ;
           compte invoc = anc num invoc ;
           base display = anc bd ;
           link bloc13 to basebloc ;
           unstack bloc13 from sysarea ;
           GOTO retour épilogue ;
         dehors :
```

D.13 - Procédure -

```
P : P2 : P3 : PROC (X, Y) RETURNS (PTR)
    => P : PROC (X, Y) RETURNS (PTR) ;
       P2 : P3 : ENTRY (X, Y) RETURNS (PTR)

P : PROC (X) RETURNS (BIT(1)) ; <corps>  ; END
    => GOTO Pfin ;
       BEGIN ;
       P : ENTRY (X) RETURNS (BIT(1)) ;(la portée de P est résolue
           <corps> ;                    avant transformation)
```

```
        RETURN ;
        END ;
    Pfin :
```

D.14 - Entry -

```
E1 : E2 : ENTRY (X)
   => E1 : ENTRY (X) ;
      E2 : ENTRY (X) ;

E : ENTRY (Y, Z) RETURNS (FIXED)
   => GOTO Efin ;
      DCL 1 Eparam BASED (adrarg),
         2 résultat3 FIXED,     (pour le 3ème point d'entrée du bloc)
         2 Yadr PTR ;
         2 Zadr PTR ;
      link Y to bloc13.Yadr ;  (s'il s'agit du 13ème bloc encontré)
      link Z to bloc13.Zadr ;
      retour prolog =Eprochaine ;
      GOTO prolog13 ;          (le prologue établit le bloc AUTOMATIC)
Eprochaine : entrée13 = 3 ;
      bloc13 = Eparam, BY NAME ;
Efin : ;
      DCL (résultat3 FIXED,        déclarations ajoutées
           entrée13 FIXED) AUTO ;  au bloc AUTOMATIC
```

D.15 - Return -

```
RETURN ; => retour epilog = bloc13 . adrretour ; (pour 13ème bloc)
            GOTO epilog 13 ;
RETURN (A + B) ;
         => DCL aig(5) LABEL ; (si le bloc a 5 points d'entrées)
            GOTO aig (entrée 13) ;
       (aig(1) : résultat1 = A + B ; GOTO prochaine ;
       (aig(2) : résultat2 = A + B ; GOTO prochaine ;
                          ...
       (aig(5) : résultat5 = A + B ;
       prochaine : RETURN ;
```

D.16 - Call -

```
CALL G(X) => CALL P(X)
     (pour DCL G GENERIC (P ENTRY(FIXED), Q ENTRY(FLOAT))
        et DCL X FIXED)

CALL P(X, Y+Z, 1, S, SIN(Y))
     (pour DCL P ENTRY(FIXED, ... FIXED) ;
        et DCL X FIXED, S CHAR(10) ;)
=>   DCL (t1, t2, t3, t4) FIXED ;
     t1 = Y + Z ;
     t2 = 1 ;
     t3 = S ;
     t4 = SIN(Y) ;
     CALL P (X, t1, t2, t3, t4)

CALL P(X, Y)
=>   DCL 1 args AUTO,
         2 (Xadr, Yadr) PTR ;
     Xadr = ADDR(X) ;
     Yadr = ADDR(Y) ;
     adrarg = ADDR(args) ;
     adrretour = prochaine ;
     GOTO P ;
prochaine : Z = résultat ;
```

D.17 - Utilisation de fonction -

```
Z = G(X) ; => Z = P(X) ;    (comme pour GENERIC dans CALL)
Z = P(X, Y + Z) ;           (comme pour les "dummies" dans CALL mais
                             avec RETURNS(FLOAT) et
=>   DCL t1 FIXED ;          DCL Z COMPLEX)
     DCL t2 FLOAT ;
     t1 = Y + Z ;
     t2 = P(X, t1) ;
     Z = t2 ;
```

```
Z = P(X, Y) ;
=> DCL 1 args AUTO,
       2 résultats COMPLEX
       2 (Xadr, Yadr) PTR ;
   Xadr = ADDR(X) ;
   Yadr = ADDR(Y) ;
   adrarg = ADDR(args) ;
   adrretour = prochaine ;
   GOTO P ;
prochaine : Z = résultat ;
```

D.18 - GO TO -

```
GOTO L => goto L          (étiquette constante)
GOTO V => goto Vvaleur    (étiquette variable dans le même bloc)
GOTO V                    (étiquette variable hors du bloc)
=> boucle : IF V num invoc = num invoc courant
            THEN goto Vvaleur ;
            ELSE DO ;
                 retour épilogue = boucle ;
                 goto épicourant ;
            END
```

D.19 - IF -

```
IF E THEN GOTO L ;
=> t = E ;                (si L est constante,
   go-if t to L ;         sinon analogue au GO TO)

IF E THEN DO ; <listeinstr> ; END
=> IF E ¬= 0 THEN GOTO t ; <listeinstr> ; t :
```

```
IF E THEN DO ; <listeinstr> ; END ;
     ELSE<instr>
=> IF E ¬ = 0 THEN GOTO t1 ;
   <listeinstr> ;
    GOTO t2 ;
t1 : <instr> ;
t2 :
```

D.2 - Déclarations -

D.21 - Prélude -

```
layout display
      sketch adrniveau1 : a, 1
      sketch adrniveau2 : a, 1          un pour chaque niveau
        ...                             statique du programme donné
      sketch adrniveau n : a, 1
frame
link display to base display
```

D.22 - Types de données -

```
FIXED BINARY(20, 3) => xb 20 + 3
FIXED DECIMAL(5, -2) => xd 5 - 2
FLOAT BINARY(13) => fb 13
FLOAT DECIMAL(5)  => fd 5

DCL C COMPLEX FIXED BINARY(15, 0)
=> DCL 1 C, 2(réel, imag) FIXED BINARY(15, 0)
   (analogue pour les autres types COMPLEX)

DCL P PICTURE '999V99'
=> DCL 1 P,
     2 Pforme CHAR(6) INITIAL ('999V99'),
     2 Pvaluer FIXED DEC (5, 2) ;
```

```
POINTER => a                    (Champ attribut)

DCL C CHAR (N) VARYING ; (même pour BIT)
=> DCL 1 C info
     2 Cadr PTR,
     2 Cmax FIXED BINARY(15),
     2 Clongueur FIXED BINARY(15) ;
   describe C @ Cadr, 0 : c, Cmax ;
   Cmax = N ;

DCL C CHAR (N) ;
=> DCL 1 Cinfo,
     2 Cadr PTR,
     2 Clongueur FIXED BIN(15) ;
   describe C @ Cadr, 0 : c, Clongueur ;
   Clongueur = N ;

DCL L LABEL
=> layout L info : 1 ;
        sketch L valeur : l, 1 ;
        sketch L num invoc : xb, 1 ;
   frame

DCL O OFFSET (A)
=> sketch O : xb, 1 ;
   describe O ptr @ A adr, O : a, 1

DCL A AREA (N)
=> DCL 1 A info,
     2 A adr PTR,
     2 A compte FIXED,        (compteur d'allocations)
     2 A taille FIXED,        (taille maximum)
     2 A taillecour FIXED,    (taille courante)
describe A @ Aadr, O : s, Ataille ;
Ataille = N ;
```

```
DCL F FILE OUTPUT PRINT ;
=> sketch F : attr protofile, 1 ;
   value output = 1 ;              ("output" et "print" sont définis
   value print = 1 ;                dans "protofile")
   file F ;
```

D.23 - Structures -

```
DCL 1 S, 2 T, 3 X, 3 Y, 2 U ;
=> layout S ;
      layout T ;
            sketch X ;
            sketch Y ;
      frame ;
      sketch U ;
   frame ;
```

D.24 - Tableaux -

```
DCL A( M : N, P : Q) FLOAT ;
=> layout A info : 1 ;                    ("dope vector")
         sketch bi 1 : xb 15+0, 1 ;
         sketch bs 1 : xb 15+0, 1 ;       (bornes inférieures et supérieures
         sketch enj 1 : xb 31+0, 1 ;       et l'enjambée pour chaque dimen-
         ...                               sion)
         sketch dv : xb 31+0, 1 ;         (déplacement virtuel)
         sketch Aadr : a, 1 ;             (l'adresse de contenu)
   frame ;
   describe A @ Aadr : a, 1
   bi1 = M ; bs1 = N ; bi2 = P ; bs = Q ;
   enj2 = bs2 - bi2 + 1 ; enj1 = enj2 * (bs1 - bi1 + 1) ;
   dv = - (taille A * bi1 + enj1 * bi2) ;

DCL A(N) => DCL A (1 : n)

DCL A(*) => (comme auparavant mais sans les instructions actives)
```

D.25 - Types des emplacements mémoires -

```
DCL X STATIC EXTERNAL
=> share X as "X"

DCL X STATIC INTERNAL
=> fasten X

DCL X AUTOMATIC
=> sketch X                (placé dans le layout pour le bloc local)

DCL X CONTROLLED INTERNAL
=> layout Xinstance ;
        sketch X ;
        sketch Xlink ;
   frame ;
   link Xinstance to Xancre ;
   fasten Xancre ;

DCL X CONTROLLED EXTERNAL
=> (comme auparavant sauf que le fasten devient un task-use)

DCL X BASED (P)
=> link X to P
```

D.26 - Valeurs initiales -

```
X INITIAL CALL P
=> CALL P                  (à l'intérieur du prologue)

X INITIAL ('XYZ')
=> value X = 'XYZ'
```

```
X (100) INITIAL ((25) 0, (25) *, (25) 1)
=> layout X initial : 1 ;
        sketch X i1 : ..., 25 ; value X i1 = 0 ;
        sketch X i2 : ..., 25 ;
        sketch X i3 : ..., 25 ; value X i3 = 1 ;
   frame ;
   value X = Xinitial
```

D.3 - Affectations -

D.31 - Structures -

```
S = T + U ;
=> S.W = T.X + U.Y ;                (normal)
   S.X = T.W + U.Z ;
   S.Y = T.Y + U.X ;

S = T + U, BY NAME ;
=> S.X = T.X + U.X ;                (par nom)
   S.Y = T.Y + U.Y ;
```

D.32 - Tableaux -

```
A = B + C ; => A(*, *) = B(*) + C(*, *, *) ;      (sans indices)

A(1, *) = B(*) + C(*, 2, 3) ;                     ("cross-sections")
=> DO i = LBOUND (A, 2) TO HBOUND (A, 2) ;
     A(1, i) = B(i) + C(i, 2, 3) ;
   END ;

I = LENGTH(5) ;                                   ("built-ins")
=> DO i = LBOUND (I, 1) TO HBOUND (I, 1) ;
     I(i) = LENGTH (S(i)) ;
   END ;
```

D.33 - Partie gauche multiple -

```
A, B, C = D => t = D ; A = t ; B = t ; C = t ;
```

D.34 - Défauts pour les qualificateurs -

```
A = A + 1
  => P -> A = P -> A + 1          (si DCL A BASED (P))

A = A + 1
  => S.A = S.A + 1                (si DCL 1 S, 2 A)
```

D.35 - Fonctions -

```
Z = F(X) + 1 => t = F(X) ; Z = t + 1
```

D.36 - Expressions -

```
A = B * C + D * E
  => t1 = B * C ; t2 = D * E ; A = t1 + t2
                  (ils ont la même forme que les opérateurs BL/1)
```

D.37 - Indexage -

```
A(I) = B(J)
=> t = I * taille A ;
   describe A elem @ adr A, t : attr A, 1 ;
   t2 = J * taille B ;
   describe B elem @ adr B, t2 : attr B, 1 ;
   A elem = B elem ;

A(I, J, K)
=> A(I * enj2 + J * enj1 + K * taille A)
```

D.38 - Pointeurs de qualification -

```
P -> A = 1
  => describe t @ P, off A : attr A, rep A ;
     t = 1
```

D.39 - Conversions implicites -

```
C = C + 1                              (avec DCL C CHAR (...))
  => t = car_à_fix (C) ;
     t = t + 1 ;
     C = fix_à_car (t)
```

D.3A - Opérateurs non primitifs -

```
Z = 1.0 I * Z
  => t.réel = 0. * Z.réel + 1.0 * Z.imag ;
     t.imag = 1.0 * Z.réel + 0. * Z.imag ;
     Z.réel = t.réel ;
     Z.imag = t.imag ;
```

D.3B - OFFSET -

```
0 -> A = 1
  => describe t @ 0 pointeur, off A : attr A, rep A ;
     t = 1
```

D.3C - Etiquettes -

```
V = L                                  (L constante, V variable)
  => Vvaleur = L ;
     Vnum invoc = num invoc courant

V = V2                                 (V et V2 variable)
  => Vvaleur = V2valeur ;
     Vnum invoc = V2 num invoc
```

D.4 - Multi-tâche -

D.41 - Déclarations pour la tâche -

```
layout données tâche ;
      sketch base display : a, 1 ;          (adresse de base de display)
      sketch adrretour : 1, 1 ;             (adresse de retour de procédure)
      sketch retour prologue : 1, 1 ;       (adresse de retour de prologue)
      sketch retour épilogue : 1, 1 ;       (adresse de retour d'épilogue)
      sketch adrarg : a, 1 ;                (adresse d'arguments de procédure)
      sketch épilogue courant : 1, 1 ;      (adresse d'épilogue courant)
      sketch num invoc courant : xb, 1 ;    (numéro d'invocation du bloc courant)
      sketch compte invoc : xb, 1 ;         (dernier numéro affecté)
frame ;
link données tâche to base tâche ;
```

D.42 - EVENT -

```
DCL E EVENT
=> DCL 1 E,
      2 E complété BIT(1) INIT('0' B),   (0 => pas complété)
      2 E status FIXED INIT(0),          (0 => complété normalement)
      2 E active BIT(1) INIT(0' B),      (sémaphore d'accès)
      2 E listewait PTR                  (liste des instructions WAIT
                                          actives)
```

D.43 - TASK -

```
DCL T TASK
=> DCL A T,
      2 T priorité,
      2 T event,                         (EVENT attaché)
      2 T épilogue                       (fin de la tâche)
```

D.44 - WAIT -

```
WAIT (E1, E2, E3) ; => WAIT (E1, E2, E3) (3) ;

WAIT (E1, E2) (1) ;
=> heap msg wait in sysheap ;      (avec un prélude qui contient
   pos wait = test ;                   DCL 1 msgwait BASED (adr msg wait) ;
   msg prochain = E1 listewait ;          2 pos wait LABEL,
   E1 listewait = adr msg wait ;          2 msg prochain PTR ;)

   heap msg wait in sysheap ;
   pos wait = test ;
   msg prochain = E2 listewait ;
   E2 listewait = adr msg wait ;

   compte = 1 ;                    (le nombre de WAIT s à attendre)
   mark drapeau
   goto prochaine

test : compte = compte - 1 ;       (les appels sont comptés)
   IF compte = 0 THEN erase drapeau ;
   goto retour test                (retour à tâche qui déverrouille)

prochaine :
```

D.45 - EXIT -

```
EXIT => goto T épilogue
```

D.46 - STOP -

```
STOP
  => SIGNAL FINISH ;
     GOTO basedetâchemajeure — > T épilogue
```

D.5 - Entrées-Sorties -

Les entrées-sorties sont presque toutes des appels de routines du système et ne sont pas explicitées ici.

E - TRANSFORMATIONS POUR LES MODIFICATIONS EN ALGOL68 -

Cet appendice est destiné à compléter la description commencée en section 5.2. On montrera simplement les transformations syntaxiques, les macros-syntaxiques équivalentes étant faciles à déduire (un grand nombre). Il y a deux sections : la première (E.1) qui décrit la traduction ALGOL68 normal vers une forme préfixée, et la deuxième (E.2) qui montre comment ajouter les opérateurs modifications.

E.1 - ALGOL68 : infixée vers préfixée -

Pour commencer, il faut que le texte ALGOL68 devienne une forme préfixée où chaque noeud est composé d'un opérateur suivi par zéro à n sous-noeuds. Dans ce qui suit les sous-noeuds seront indiqués par un soulignement (⌊_____⌋) qui fera ressortir leurs structures.

E.11 - Propositions -

(a) début train1 exit éti : train2 fin ⇒ exit train1 étiquette éti train2

(b) unité, propser ⇒ ; unité propser

(c) si cond alors propser1 sinon proposer2 fsi ⇒ si cond propser1 propser2

(d) si cond alors propser fsi ⇒ si cond propser skip

(e) cas propent dans u1, u2 sinon propser sac ⇒ casent propent vc u1 u2 propser

si l'out manque, il est remplacé par fant)

(f) cas m1, m2 {:: / ::=} prop dans u1, u2 sinon propser sac

⇒ casun vm m1 m2 {:: / ::=} prop vc u1 u2 propser

(si l'out manque, il est ramplacé par fant)

(g) (u1, u2, u3) ⇒ display vd u1 vd u2 u3

(h) pour i depuis u1 pas u2 jusqu'à u3 tantque propser faire u4 ⇒ faire i u1 u2 u3 propser u4

(toute proposition manquante est remplacée par fant)

E.12 - Déclarations -

(a) mode indicateur = déclarateur ⇒ mode indicateur déclareur listedebornes

(<listedebornes> est composé de toutes les bornes trouvées dans <déclareur>))

(b) priorité indicateur = niveau ⇒ priorité indicateur niveau

(c) déclareur identificateur = unité ⇒ dclident déclareur listedebornes identificateur unité

(<listedebornes> comme ci-dessus)

(d) déclareur identificateur $\left\{ :=_{\epsilon} \text{ unité} \right\}$ ⇒ dclvar déclareur listedebornes identificateur $\left\{ \text{unité}_{\epsilon} \right\}$

(<listedebornes> comme ci-dessus)

(e) déclaration1, déclaration2 ⇒ ; déclaration1 déclaration2

(f) déclareur i1, i2 ⇒ dclvar déclareur listedebornes vidf i1 i2 fant

(g) proc $\{^{\text{plan}}_{\epsilon}\}$ ident = unité ⇒ dclident déclareur listedebornes ident unité

(<déclareur> est le mode de la procédure et <listedebornes> est toutes les bornes contenues dans <plan>)

(h) op $\{^{\text{plan}}_{\epsilon}\}$ opérateur = unité ⇒ dclident déclareur listedebornes opérateur unité

(analogue à la procédure (g))

(i) bi1 : bs1, bi2 : bs2 ⇒ lb bi1 lb bs1 lb bi2 bs2

E.13 - Confrontations -

(a) tertiaire := unité ⇒ := tertiaire unité

(b) tertiaire1 { :: / ::= / :=: / :≠: } tertiaire2 ⇒ { :: / ::= / :=: / :≠: } tertiaire1 tertiaire2

(c) déclareur : unité ⇒ forceur déclareur unité

E.14 - Formules -

(a) opérateurmonadique secondaire ⇒ opm opérateurmonadique secondaire

(b) sec1 opérateurdyadique sec2 ⇒opd opérateurdyadique sec1 sec2

E.15 - Cohésions -

(a) sélecteur de secondaire ⇒ de sélecteur secondaire

(b) loc déclareur ⇒ loc déclareur listedebornes

(générateur) (<listedebornes> est composé de toutes les bornes trouvées dans <déclareur>)

(c) $\left\{\underset{\epsilon}{\text{tas}}\right\}$ déclareur = tas déclareur listedebornes

(comme ci-dessus)

E.16 - Bases -

(a) (propser) ⇒ propser

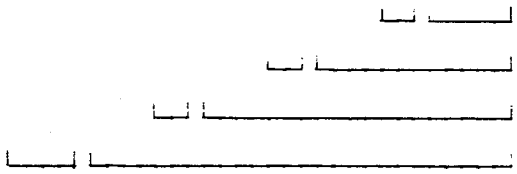
(b) {prim (tert1, tert2, tert3) / prim [tert1, tert2, tert3]} ⇒ ta prim vta tert1 vta tert2 vta tert3 finta

(tranche)

(c) t1 : t2 @ t3, reste ⇒ massicot t1 t2 t3 reste

(massicot au lieu d'indice en dessus)

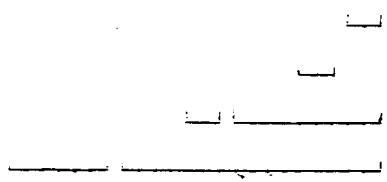
(d) prim (u1, u2, u3) => ta prim vta u1 vta u2 vta u3 finta
(appel)



(e) ident => idf ident

(f) 3.14 => notation 3.14

(g) $ format $ => $ format lrd u1 lrd u2 u3



(le noeud brd représente toutes les répétitions dynamiques (u1, u2, ...) qui se trouvent dans <format>)

(h) fant => fant

(i) nil => nil

(j) ϵ => vide (est introduit dans les positions où un rang de zéro élément est indiqué par la syntaxe)

(k) allera éti => idf éti

(1) ((paramètres formels) déclareur : unité) => routine modeproc listedebornes forceur déclareur unité

(notation de routine : <modeproc> est le mode de procédure délivré, <listedebornes> est la liste de toutes les bornes spécifiées en <paramètres formels>).

E.2 - ALGOL68 : préfixée vers préfixée modifiée -

Une fois le code préfixé produit (utilisant les transformations de la section précédente), les modifications peuvent être déterminées par macros de la façon décrite en section 5.2. Cette section va compléter la description avec la notation abrégée des transformations syntaxiques.

Un noeud a la forme :

$$\underline{\text{mot-clé}}\ \text{sous-noeud}_1\ \ldots\ \text{sous-noeud}_n$$

Les paramètres (modes et contextes) transmis vers le bas de l'arbre syntaxique sont indiqués par un triangle inversé (∇) :

$$(\nabla\ c\ \ m)$$

La lettre c dénote un contexte (fort, ferme, faible, mou ou vide). La lettre m dénote un mode (y compris : "inconnu").

Les paramètres qui restent à un certain niveau sont marqués par un carré ($\square$) et ceux qui remontent dans l'arbre par un triangle (Δ). Dans tous cas, le groupe de paramètres est parenthésé et se trouve à gauche d'un noeud ou d'un sous-noeud.

Notons que quand les paramètres remontent,la forme normale comprend à la fois les modes à priori et à postériori :

$$(\Delta\ c\ \ m_{post}\ \ m_{pre})$$

Après la macro de la section 5.231, cette dernière forme est une des transformations suivantes :

$(\Delta\ \underline{\text{fort}}\ m_1\ m_2) \Rightarrow (\underline{\text{fort}}\ m_1\ m_2) \overset{*}{\Rightarrow} \varepsilon$

$(\Delta\ \underline{\text{ferme}}\ m_1\ m_2) \Rightarrow (\underline{\text{ferme}}\ m_1\ m_2) \overset{*}{\Rightarrow} \varepsilon$

$(\Delta\ \underline{\text{faible}}\ m_1\ m_2) \Rightarrow (\underline{\text{faible}}\ m_1) \overset{*}{\Rightarrow} \underline{\text{terminé}}\ m_2$

$(\Delta\ \underline{\text{mou}}\ m_1\ m_2) \Rightarrow (\underline{\text{mou}}\ m_1) \overset{*}{\Rightarrow} \underline{\text{terminé}}\ m_2$

$(\Delta\ \underline{\text{vide}}\ m_1\ m_1) \Rightarrow (\underline{\text{vide}}\ m_1) \overset{*}{\Rightarrow} \underline{\text{terminé}}\ m_1$

Les transformations ci-dessus indiquées par le symbole ($\overset{*}{\Rightarrow}$) sont celles qui résultent de l'algorithme de "liste de modifications" (5.22).

Les noeuds sont abrégés par la lettre "n".

La transformation d'un programme écrit en ALGOL68 commence avec la phrase

(∇ fort neutre) programme

et termine quand tous les paramètres entre parenthèses ont disparu du texte préfixé.

Si une transformation dépend de la valeur vrai d'un prédicat, ce dernier est écrit après la transformation entre crochets ([ ]) :

phrase originelle ⇒ phrase de remplacement [prédicat]

est équivalente à :

noeud → phrase originelle

= si prédicat alors phrase de remplacement fsi

(1) acheveur -

$(\nabla\ c\ m)$ exit $n_g\ n_d \Rightarrow$ exit $(\nabla\ c\ m)$ équilibrer $n_g\ n_d$

exit (terminé m) équilibrer $n_g\ n_d \Rightarrow$ (terminé m) exit $n_g\ n_d$

(2) équilibrage -

$(\nabla$ fort m) équilibrer $n_g\ n_d \Rightarrow (\nabla$ fort m) n_g $(\nabla$ fort m) n_d

$(\nabla$ ferme m) équilibrer $n_g\ n_d \Rightarrow$ { $(\nabla$ ferme m) n_g $(\nabla$ fort m) n_d ; $(\nabla$ fort m) n_g $(\nabla$ ferme m) n_d

$(\nabla\ c\ m)$ équilibrer $n_g\ n_d \Rightarrow$ { équilibrer $(\nabla\ c\ m)\ n_g\ n_d$; équilibrer n_g $(\nabla\ c\ m)\ n_d$ $\quad [c = \{$ faible, mou, vide $\}]$

équilibrer (terminé m) $n_g\ n_d \Rightarrow$ (terminé m) $n_g\ n_d$

équilibrer n_g (terminé m) $n_d \Rightarrow$ (terminé m) $n_g\ n_d$

(3) continuer -

$(\nabla\ c\ m)\ ;\ n_g\ n_d \Rightarrow ;\ (\nabla$ fort neutre) n_g $(\nabla\ c\ m)\ n_d$

$;\ n_g$ (terminé m_d) $n_d \Rightarrow$ (terminé m_d) $;\ n_g\ n_d$

(4) proposition si -

$(\nabla\ c\ m)$ si $n_c\ n_a\ n_s \Rightarrow$ si $(\nabla$ fort bool) n_c $(\nabla\ c\ m)$ équilibrer $n_a\ n_s$

si n_c (terminé m) $n_a\ n_s \Rightarrow$ (terminé m) si $n_c\ n_a\ n_s$

(5) proposition cas entier -

$(\nabla\ c\ m)$ casent $n_c\ n_d\ n_s \Rightarrow$ casent $(\nabla$ fort ent) n_c $(\nabla\ c\ m)$ équilibrer $n_d\ n_s$

casent n_c (terminé m) $n_d\ n_s \Rightarrow$ (terminé m) casent $n_c\ n_d\ n_s$

(6) virgule de cas -

(∇ c m) vc n_g n_d ⇒ vc (∇ c m) équilibrer n_g n_d

vc (terminé m) n_g n_d ⇒ (terminé m) vc n_g n_d

(7) proposition cas de conformité -

(∇ c m) casun n_1 n_o n_u n_d n_a ⇒ casun (∇ mou inconnu) n_1 n_o (∇ vide inconnu) n_1 (∇ c m) équilibrer n_d n_a

casun (terminé m) n_1 ⇒ casun n_1

casun n_1 n_o (terminé m) n_u ⇒ casun n_1 n_o n_u

casun n_1 n_o n_u (terminé m) n_d n_a ⇒ (terminé m) casun n_1 n_o n_u n_d n_a

(8) virgule de liste de conformité -

(∇ mou inconnu) vm n_g n_d ⇒ vm (∇ mou inconnu) n_g (∇ mou inconnu) n_d

vm (terminé m) n_g ⇒ vm n_g

vm n_g (terminé m) n_d ⇒ vm n_g n_d

(9) proposition collatérale (display) -

(∇ fort neutre) display n ⇒ display (∇ fort neutre) n

(∇ fort struct (listedemodes)) display n ⇒ display (∇ fort listedemodes) n

(∇ c rang m) display n ⇒ display (∇ c m) n [c = fort ∨ c = ferme]

(10) virgule display -

(∇ fort neutre) vd n_g n_d ⇒ vd (∇ fort neutre) n_g (∇ fort neutre) n_d

(∇ fort lm) vd n_g n_d ⇒ vd (∇ fort m) n_g (∇ fort lm_2) n_d [lm → m, lm_2 ∧ lm_2 ≠ m_2]

vd (∇ fort m) n_g (∇ fort m_2) n_d [lm → m, lm_2 ∧ lm_2 → m_2]

(∇ c m) vd n_g n_d ⇒ vd (∇ c m) n_g (∇ c m) n_d [c = fort ∨ c = ferme]

(11) proposition faire -

(∇ fort neutre) faire n_i n_d n_p n_j n_t n_f

⇒ faire n_i (∇ fort ent) n_d (∇ fort ent) n_p (∇ fort ent) n_j (∇ fort bool) n_t (∇ fort neutre) n_f

(12) étiquette -

(∇ c m) étiquette n_c n_r ⇒ étiquette n_c (∇ c m) n_r

étiquette n_c (terminé m) n_r ⇒ (terminé m) étiquette n_c n_r

(13) déclaration de mode -

(∇ fort neutre) mode n_i n_d n_l ⇒ mode n_i n_d (∇ fort ent) n_l

(14) déclaration de priorité -

(∇ fort neutre) priorité n_i n_n ⇒ priorité n_i n_n

(15) déclaration d'identité -

(∇ fort neutre) dclident n_d n_l n_i n_u ⇒ dclident n_d (∇ fort ent) n_l n_i (∇ fort n_d) n_u

(16) déclaration de variable -

(∇ fort neutre) dclvar n_d n_l n_i n_u ⇒ dclvar n_d (∇ fort ent) n_l n_i (∇ fort m) n_u [m = réf n_d]

(17) liste de bornes -

(∇ fort ent) lb n_g n_d ⇒ lb (∇ fort ent) n_g (∇ fort ent) n_d

(18) affectation -

$(\nabla\ c\ m) := n_g\ n_d \Rightarrow (\square\ c\ m) := (\nabla\ \underline{mou}\ m)\ n_g\ n_d$

$(\square\ c\ m) := (\underline{terminé}\ m_g)\ n_g\ n_d \Rightarrow (\Delta\ c\ m\ m_g) := n_g\ (\nabla\ \underline{fort}\ m_d)\ n_d \quad [m_g = \underline{réf}\ m_d]$

(19) relation de conformité -

$$(\nabla\ c\ m) \left\{\begin{matrix}::\\::=\end{matrix}\right\} n_g\ n_d \Rightarrow (\Delta\ c\ m\ \underline{bool}) \left\{\begin{matrix}::\\::=\end{matrix}\right\} (\nabla\ \underline{mou}\ \underline{inconnu})\ n_g\ (\nabla\ \underline{vide}\ \underline{inconnu})\ n_d$$

$$\left\{\begin{matrix}::\\::=\end{matrix}\right\} (\underline{terminé}\ m_g)\ n_g\ (\underline{terminé}\ m_d)\ n_d \Rightarrow \left\{\begin{matrix}::\\::=\end{matrix}\right\} n_g\ n_d$$

(20) relation d'identité -

$$(\nabla\ c\ m) \left\{\begin{matrix}:=:\\:\neq:\end{matrix}\right\} n_g\ n_d \Rightarrow (\nabla\ c\ m\ \underline{bool}) \left\{\begin{matrix}:=:\\:\neq:\end{matrix}\right\} (\nabla\ \underline{mou}\ \underline{inconnu})\ \underline{équilibrer}\ n_g\ n_d$$

$$\left\{\begin{matrix}:=:\\:\neq:\end{matrix}\right\} (\underline{terminé}\ m)\ n_g\ n_d \Rightarrow \left\{\begin{matrix}:=:\\:\neq:\end{matrix}\right\} n_g\ n_d$$

(21) forceur -

$(\nabla\ c\ m)\ \underline{forceur}\ n_d\ n_u \Rightarrow (\Delta\ c\ m\ n_d)\ \underline{forceur}\ n_d\ (\nabla\ \underline{fort}\ n_d)\ n_u$

(22) opérateur monadique -

$(\nabla\ c\ m)\ \underline{opm}\ n_o\ n_s \Rightarrow \{(\Delta\ c\ m\ m_r^{[i]})\ \underline{opm}\ n_o\ (\nabla\ \underline{ferme}\ m_s^{[i]})\ n_s\}_{i=1}^{l}$

[$m^i = \underline{proc}\ (m_s^{[i]})\ m_r^{[i]}$ où les $m^{[i]}$ sont tous les modes définis pour l'opérateur n_o.

Ils sont essayés l'un après l'autre dans l'ordre: le plus intérieur vers l'extérieur.

cf. section 5.24]

(23) opérateur dyadique -

$(\nabla\ c\ m)\ \underline{opd}\ n_o\ n_g\ n_d\ ; \Rightarrow \{(\Delta\ c\ m\ m_r^{[i]})\ \underline{opd}\ n_o\ (\nabla\ \underline{ferme}\ m_g^{[i]})\ n_g\ (\nabla\ \underline{ferme}\ m_d^{[i]})\ n_d\}_{i=1}^{l}$

[le même que ci-dessus sauf que $m^{i} = \underline{proc}\ (m_g^{[i]}, m_d^{[i]})\ m_r^{[i]}$]

(24) générateur -

$(\nabla\ c\ m) \left\{\begin{matrix}\underline{loc}\\ \underline{tas}\end{matrix}\right\} n_d\ n_l \Rightarrow (\Delta\ c\ m\ m_g) \left\{\begin{matrix}\underline{loc}\\ \underline{tas}\end{matrix}\right\} n_d\ (\nabla\ \underline{fort}\ \underline{ent})\ n_l \quad [m_g = \underline{réf}\ n_d]$

(25) sélection -

$(\nabla\ c\ m)\ \underline{de}\ n_g\ n_d \Rightarrow (\square\ c\ m)\ \underline{de}\ n_g\ (\nabla\ \underline{faible}\ \underline{inconnu})\ n_d$

$(\square\ c\ m)\ \underline{de}\ n_g\ (\underline{terminé}\ m_d)\ n_d \Rightarrow (\Delta\ c\ m\ m_s)\ \underline{de}\ n_g\ n_d$

$[m_d = \underline{struct}\ (\ldots, m_s\ n_g, \ldots)]$

$(\square\ c\ m)\ \underline{de}\ n_g\ (\underline{terminé}\ m_d)\ n_d \Rightarrow (\Delta\ c\ m\ \underline{réf}\ m_s)\ \underline{de}\ n_g\ n_d$

$[m_d = \underline{réf}\ \underline{struct}\ (\ldots, m_s\ n_g, \ldots)]$

(26) tranche-appel -

$(\nabla\ c\ m)\ \underline{ta}\ n_g\ n_d \Rightarrow \begin{cases}(\square\ c\ m)\ \underline{tranche}\ (\nabla\ \underline{faible}\ \underline{inconnu})\ n_g\ n_d & [m = \ldots\ \underline{rang}\ m_r]\\ (\Delta\ c\ m\ m_r)\ \underline{appel}\ (\nabla\ \underline{ferme}\ m_p)\ n_g\ (\triangledown\ lm)\ n_d & [m = \ldots\ m_p \wedge m_p = \underline{proc}\ (lm)\ m_r]\end{cases}$

$(\square\ c\ m)\ \underline{tranche}\ (\underline{terminé}\ m_g)\ n_g\ n_d \Rightarrow (\boxed{2}\ c\ m)\ \underline{tranche}\ n_g\ (\triangledown\ m_g)\ n_d$

$(\boxed{2}\ c\ m)\ \underline{tranche}\ n_g\ (\triangle\ m_d)\ n_d \Rightarrow (\Delta\ c\ m\ m_d)\ \underline{tranche}\ n_g\ n_d$

(27) virgule de tranche-appel -

($\overline{\text{s}}$ m) vta n_g n_d ⇒ vta (∇ fort ent) n_g ($\overline{\text{s}}$ m_d) n_d [m = rang m_d]

vta n_g ($\underline{\text{s}}$ m_d) n_d ⇒ ($\underline{\text{s}}$ m_d) vta n_g n_d

($\overline{\text{c}}$ ml) vta n_g n_d ⇒ vta (∇ fort m) n_g ($\overline{\text{c}}$ lm_2) n_d [lm → m, lm_2]

(28) fin de tranche-appel -

($\overline{\text{s}}$ m) finta ⇒ ($\underline{\text{s}}$ m) finta

($\overline{\text{c}}$ lm) finta ⇒ finta

(29) massicot -

($\overline{\text{s}}$ m) massicot n_i n_s n_n n_r ⇒ massicot (∇ fort ent) n_i (∇ fort ent) n_s (∇ fort ent) n_n ($\overline{\text{s}}$ m) n_r

massicot n_i n_s n_n ($\underline{\text{s}}$ m) n_r ⇒ ($\underline{\text{s}}$ m) massicot n_i n_s n_n n_r

(30) identificateur -

(∇ c m) idf n_i ⇒ (Δ c m m_i) idf n_i [m_i est le mode à priori de l'identificateur]

(31) notation (numérique) -

(∇ c m) notation n_n ⇒ (Δ c m m_n) notation n_i [m_n est le mode à priori de la notation]

(32) format -

(∇ c m) $ n_p n_l ⇒ (Δ c m format) $ n_p (∇ fort ent) n_l

(33) liste de répétitions dynamiques -

(∇ fort ent) lrd n_g n_d ⇒ lrd (∇ fort ent) n_g (∇ fort ent) n_g

(34) fantôme -

(∇ fort m) fant ⇒ fant

(35) nil -

(∇ fort m) nil ⇒ nil [m = réf ...]

(36) vide -

(∇ fort m) vide ⇒ vide [m = rang ...]

(37) notation de routine -

(∇ c m) routine n_m n_l n_u ⇒ (Δ c m n_m) routine n_m (∇ fort ent) n_l (∇ fort n_m) n_u

BIBLIOGRAPHIE

[1] - V. BAJAR -
"Etude de la compilation de Basel, langage de la famille ALGOL 68",
Thèse, Université Scientifique et Médicale de Grenoble,
Février 1973 - GRENOBLE -

[2] - D. BERT -
"Etude d'éléments fondamentaux des langages de programmation",
Thèse, Université Scientifique et Médicale de Grenoble,
Mai 1973 - GRENOBLE -

[2a] - M. BERTHAUD, D. CLAUZEL, M. JACOLIN -
Grenoble System Language - Etude n° FF2.0133,
Centre Scientifique IBM de Grenoble,
Janvier 1973 - GRENOBLE -

[3] - Ph. CHATELIN -
"Un analyseur de grammaire hors-contexte",
Séminaire I.M.A.G.,
Mars 1972 - GRENOBLE -

[4] - Ph. CHATELIN, B. WILLIS -
"Présentation d'un macroprocesseur syntaxique",
Séminaire I.R.I.A.,
AVRIL 1973 - ROQUENCOURT -

[5] - P. COUSOT -
"Définition transformationnelle et implantation de langages de programmation",
Thèse, Université Scientifique et Médicale de Grenoble,
A paraître - GRENOBLE -

[6] - P. COUSOT -
"Un analyseur syntaxique pour grammaires hors contexte ascendant sélectif et général",
Congrès A.F.C.E.T.,
Novembre 1972 - GRENOBLE -

[7] - P.Y. CUNIN, M. SIMONET, B. WILLIS -
"Architecture d'un compilateur ALGOL 68 en trois passages",
Congrès A.F.C.E.T.,
Novembre 1972 - GRENOBLE -

[8] - P.Y. CUNIN, M. DELAUNAY, M. SIMONET, J. VOIRON, B. WILLIS -
"Modifications en ALGOL 68 - Implantation possible",
Journée ALGOL 68, I.P.P.,
AVRIL 1973 - PARIS -

[9] - J. EARLEY -
"An efficient context-free parsing algorithm",
C.A.C.M. v.13,
(Février 1970)

[10] - FOSTER -
"A syntax improving program",
Computer Journal v. 11, *(1968)*

[11] - J. GARWICK -
"GPL, a truly general programming language",
C.A.C.M. v. 11,
(Septembre 1968)

[12] - D. GRIES -
"Compiler construction for digital computers",
Wiley,
1971 - NEW-YORK -

[13] - I. B. M. -
"PL/1 (F) Compiler, Programming Logic Manual",
Form Y28 - 6800,

[14] - E. IRONS -
"A syntax directed compiler for Algol 60",
C.A.C.M., v. 4,
(Janvier 1961)

[15] - Ph. JORRAND, S. SCHUMAN -
"Spécification des langages de programmation et leurs traducteurs au moyen de macros-syntaxiques",
Congrès A.F.C.E.T., *1970 - PARIS -*

[16] - LANDIN -
"The mechanical evaluation of expressions",
Computer Journal v.6,
(1963)

[17] - N. Mc KEEMAN et AL -
"A compiler generator implemented for the IBM system/360",
Prentice Hall,
(1970 - NEW-JERSEY)

[18] - P. NAUR et AL -
"Revised report on the algorithmic language ALGOL 60",
C.A.C.M., v.6
(Janvier 1963)

[19] - P. POOLE -
Séminaire I.M.A.G.,
1972 - GRENOBLE -

[20] - S. SCHUMAN, Ph. JORRAND -
"Definition mechanisms in extensible programming languages",
Fall Joint Computer Conference,
1970 - HOUSTON (Texas)

[21] - D. SERAIN -
"Implementation de BL/1 sur 360"
Projet D.E.A.,
1972

[22] - P. UZGALIS -
"Language changes incorporated in the revised Algol 68 report", (avant-projet),
Juillet 1973

[23] - A. VAN WIJNGAARDEN et AL -
"Report on the algorithmic language Algol 68",
Mathematisch Centrum,
Octobre 1969 - AMSTERDAM -

[24] - A. VAN WIJNGAARDEN et AL -
"Draft revised report on the algorithmic language Algol 68",
Juillet 1973 - AMSTERDAM -

[25] - P. WEGNER -
"The Vienna Definition Language",
Computing Surveys,
Mars 1972

[26] - B. WILLIS -
"Implementierung PL/1 durch syntaktischen Makros",
Séminaire Techniche Universität,
Novembre 1972 - BERLIN -

[27] - B. WILLIS -
"A base language for PL/1",
Note technique I.M.A.G.,
Novembre 1970 - GRENOBLE -

[28] - B. WILLIS -
"PL/1 in an extensible language environment",
Note technique I.M.A.G.,
1969 - GRENOBLE -

[29] - B. WILLIS, Ph. CHATELIN -
"Defining and implementing PL/1",
International Computing Symposium 1973,
Septembre 1973 - DAVOS (Suisse)

[30] - B. WILLIS, Ph. CHATELIN -
"Etude et évaluation de la réalisation de PL/1 comme extension à partir d'un langage de base",
Rapport du C.R.I., Contrat 70107,
Septembre 1972 - GRENOBLE -

[31] - N. WIRTH -
"PL 360, a programming language for the 360 computers",
J.A.C.M., v. 15,
(Janvier 1968)

[32] - N. WIRTH -
"A contribution to the development of Algol",
C.A.C.M., v.9
(Juin 1966)