Méthodologie d'écriture de compilateurs - une expérience du langage ALGOL 68

Pierre-Yves Cunin, Michel Simonet, Jacques Voiron

THESE présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE

DOCTEUR INGENIEUR

Pierre-Yves Cunin

Michel Simonet

Jacques Voiron

METHODOLOGIE D'ECRITURE DE COMPILATEURS

UNE EXPERIENCE DU LANGAGE ALGOL 68

Thèse soutenue le 22 avril 1976 devant la Commission d'Examen :

Président	:	Monsieur	J. KUNTZMANN
Rapporteur	:	Monsieur	M. GRIFFITHS
Examinateurs	:	Messieurs	J.C. BOUSSARD
			I. CURRIE
			P. JORRAND
			C.H.A. KOSTER
			G. VEILLON
			J.P. VERJUS

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

M. Michel SOUTIF : Président
M. Gabriel CAU : Vice-président

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	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
	BEAUDOING André	Clinique de Pédiatrie et Puériculture
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BEZES Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Clinique Oto-Rhino-Laryngologique
	CHATEAU Robert	Thérapeutique (Neurologie)
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	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
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie minérale
	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 neurologique

MM.	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques Pures
	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	Mathématiques Appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mlle	LUTZ Elisabeth	Mathématiques Pures
	MALGRANGE Bernard	Mathématiques Pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Sémélologie médicale
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	MULLER Jean Michel	Thérapeutique (néphrologie)
	NEEL Louis	Physique du Solide
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale .
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique Nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
M.	VERAIN André	Physique
MM.	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique
	ZISMAN Michel	Mathématiques pures

109

PROFESSEURS ASSOCIES

MM.	BALMSKI Michel	Mathématiques appliquées
	COPPENS Philip	Physique
	CORCOS Gilles	Mécanique

MM.	CRABBE Pierre	CERMO
	DUTTON Guy	CERMAV
	GILLESPIE John	I.S.N.
	SAMPSON Joseph	Mathématiques pures

PROFESSEURS SANS CHAIRE

Mle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BERTRANDIAS Jean-Paul	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
Mme	BONNIER Jane	Chimie générale
MM.	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique
	CONTE René	Physique
	DEPASSEL Roger	Mécanique des Fluides
	GAUTHIER Yves	Sciences biologiques
	GAUTRON René	Chimie
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Méd.Préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KUHN Gérard	Physique
	LOISEAUX Jean	Physique nucléaire
	LUU-DUC-Cuong	Chimie Organique
	MAYNARD Roger	Physique du solide
	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROBERT André	Chimie papetière
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	STIEGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

47

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	AMBLARD Pierre	Dermatologie
	ARMAND Gilbert	Géographie
	ARMAND Yves	Chimie
	BARGE Michel	Neurochirurgie
	BARJOLLE Michel	M.I.A.G.
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamique

MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (IUT B)
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DELOBEL Claude	M.I.A.G.
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Gilbert	Urologie
	FAURE Jacques	Médecine légale
	FONTAINE Jean-Marc	Mathématiques Pures
	GAUTIER Robert	Chirurgie générale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques Appliquées
	GROS Yves	Physique (stag.)
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KOLODIE Lucien	Hématologie
	KRAKOWIAK Sacha	Mathématiques appliquées
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LEROY Philippe	Mathématiques
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MALLION Jean Michel	Médecine du travail
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (I.U.T. "A")
Mme	MINIER Colette	Physique
MM.	NEGRE Robert	Mécanique
	NEMOZ Alain	Thermodynamique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PERRET Jean	Neurologie
	PERRIER Guy	Géophysique
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
Mme	RENAUDET Jacqueline	Bactériologie
MM.	ROBERT Jean Bernard	Chimie-Physique
	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean Claude	Chimie Générale
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

(9)

MAITRES DE CONFERENCES ASSOCIES

MM.	COLE Antony	Sciences nucléaires
	FARELL César	Mécanique
	MOORSANI Kishin	Physique

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

M.	ROCHAT Jacques	Hygiène et hydrologie

Fait à Saint Martin d'Hères, AVRIL 1975

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : M. Louis NEEL
Vice-Présidents : MM. Jean BENOIT
Lucien BONNETAIN

PROFESSEURS TITULAIRES

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

PROFESSEURS ASSOCIES

MM. RABINS Michaël	Automatique
ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM. BLIMAN Samuel	Electronique
COHEN Joseph	Electrotechnique
DURAND Francis	Métallurgie
FOULARD Claude	Automatique
LANCIA Roland	Electronique
VEILLON Gérard	Informatique fondamentale et appliquée
ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM. BOUDOURIS Georges	Radioélectricité
BOUVARD Maurice	Génie mécanique
CHARTIER Germain	Electronique
GUYOT Pierre	Chimie Minérale
IVANES Marcel	Electrotechnique
JOUBERT Jean-Claude	Physique du solide
LACOUME Jean-Louis	Géophysique
MORET Roger	Electrotechnique Nucléaire
ROBERT François	Analyse numérique
SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme SAUCIER Gabrièle	Informatique fondamentale et appliquée

CHARGE DE FONCTIONS DE MAITRE DE CONFERENCES

MM. LANCEAU François	Mathématiques Appliquées
PIERRARD Jean-Marie	Hydraulique

.../...

CHERCHEURS DU C.N.R.S.

M. FRUCHART Robert	Directeur de recherche
M. ANSARA Ibrahim	Maître de recherche
M. DRIOLE Jean	Maître de recherche
M. MATHIEU Jean-Claude	Maître de recherche
M. MUNIER Jacques	Maître de recherche

Nous tenons à remercier :

Monsieur J. KUNTZMANN qui nous a fait l'honneur de présider notre jury de thèse.

M. GRIFFITHS, l'instigateur du projet d'écriture d'un compilateur pour le langage Algol 68, qui nous a accordé sa confiance tout au long de la réalisation d'une entreprise qui paraissait audacieuse. Il a su nous encourager pour la mener à terme et son pragmatisme a été décisif pour le succès de ce projet. Lors de la rédaction de cette thèse, ses avis et ses conseils nous ont été extrêmement précieux.

J.C. BOUSSARD qui a introduit le langage Algol 68 à Grenoble, nous faisant découvrir avec humour les idées qui ont conduit à la naissance d'Algol 68 et ont bouleversé les habitudes de pensée des informaticiens. Ses critiques vigoureuses ont toujours été pour nous de la plus grande utilité.

Nous voulons aussi remercier I. CURRIE, ingénieur au Royal Radar Establishement qui, lors d'un séjour à Grenoble, nous a communiqué avec beaucoup d'enthousiasme, un peu de sa grande expérience en matière de compilation et de langages. Ses conseils ont été pour nous déterminants.

P. JORRAND, Maître de Recherche au CNRS et C.H.A. KOSTER, Professeur à l'Université de Berlin, qui ont également contribué, au sein du groupe de travail WG2.1 de l'IFIP, à la création du langage Algol 68 et à la diffusion des idées qui lui sont attachées, et qui nous ont souvent aidé à comprendre des aspects obscurs du "Rapport de Définition". Ils ont accepté de juger notre travail.

G. VEILLON et J.P. VERJUS qui se sont intéressés à nos activités et ont bien voulu participer à notre jury.

Nous remercions nos collègues de l'Université de Grenoble et les membres du groupe Algol 68 de l'AFCET, en particulier les équipes de l'Université de Rennes et du MBLE à Bruxelles, avec qui nous avons eu des discussions enrichissantes.

Nous exprimons toute notre reconnaissance à C. PAIR, Professeur à l'Université de Nancy et B. LORHO, Chercheur à l'IRIA, qui, après une lecture patiente de notre manuscrit, l'ont critiqué avec beaucoup de clairvoyance. Leurs conseils et leurs avis ont permis d'approfondir certains points et ont facilité notre rédaction.

Nous remercions tous ceux qui ont fait partie de l'équipe Algol 68 de Grenoble :

Bernard BRETAGNOLLE qui a contribué avec compétence et ténacité à la construction d'une version opérationnelle du compilateur.

Bruce WILLIS dont la richesse d'idées a conduit à des solutions originales dans le compilateur et influencé certains aspects développés dans cette thèse.

Et tout particulièrement Michel DELAUNAY qui a participé à la conception et à la réalisation du prototype et à qui nous devons l'aménagement des outils syntaxiques utilisés ; sa collaboration a été fondamentale dans la réalisation de ce projet.

Toute notre reconnaissance va également à Renée CARRE-PIERRAT et Chantal PUECH qui, avec une infinie patience et une rare compétence, ont dactylographié des textes pourtant peu attrayants et à Monsieur COLLOMB du service imprimerie de la Bibliothèque des Sciences dont l'aimable concours a permis la réalisation complète de cette thèse.

TABLE DES MATIERES

INTRODUCTION

Dans cette introduction nous situons notre travail dans le contexte du langage Algol 68 et nous présentons la démarche qui a conduit à la méthode exposée dans cette thèse.

Le langage Algol 68 a été conçu dans le cadre d'un groupe de travail de l'IFIP, le WG2.1. L'objectif de ce groupe était de concevoir un langage algorithmique de haut niveau qui pourrait avoir une diffusion internationale. Le travail du groupe ne se fit pas sans remous, car tous ses membres n'approuvaient pas son orientation. Plusieurs rapports intermédiaires ont été publiés, le langage s'appelant alors Algol X. Le dernier rapport intermédiaire est paru fin 1968 ce qui fixa le nom du langage : Algol 68. Le rapport final a été publié en février 1969 [VANWIJ-1]. Dans son introduction les auteurs présentent les principes qui les ont guidés dans la conception et la description du langage. C'est un langage conçu pour la communication d'algorithmes, leur exécution sur une large gamme de calculateurs et leur enseignement à des étudiants. Ils ont été guidés par un souci d'efficacité et de clarté et par une conception "orthogonale" du langage, c'est-à-dire par une limitation du nombre des concepts primitifs et par une systématisation de leur emploi en évitant les cas particuliers. Enfin la description du langage se veut claire et complète.

Il est difficile de juger si ces objectifs ont été atteints. Un Rapport d'Evaluation a été publié en 1970 par un groupe comprenant un éventail assez large de personnes concernées par le langage [BOU-DUBY]. Notre opinion, en tant qu'écrivains de compilateurs, sur la description du langage est qu'elle est effectivement complète mais qu'elle ne devient claire et utilisable qu'après un long et patient travail d'assimilation. La syntaxe est décrite par une grammaire à deux niveaux ou W-grammaire, du nom de A. Van Wijngaarden qui a introduit ce formalisme. En l'absence d'un guide ou d'une introduction à ce nouveau formalisme, la lecture de la syntaxe nécessite un apprentissage parfois difficile et certainement décourageant pour un lecteur isolé (une telle introduction a été réalisée depuis [CLE-UZGA]). La sémantique est décrite en anglais formalisé, sous une forme algorithmique qui s'apparente aux méthodes interprétatives. La traduction française du rapport de définition commence par une brève présentation des principaux éléments du langage [AFCET-1]. Nous aurions apprécié une telle introduction dans le rapport original, ainsi que des commentaires plus nombreux dans la description elle-même.

Ce rôle introductif a été en partie rempli par une introduction informelle à Algol 68 de Ch. Lindsey et S.G. van der Meulen [LIN-MEUL], dont une version préliminaire a été diffusée parallèlement au rapport. Ce document a été d'une aide considérable à la compréhension du langage. Il constituait alors le seul ouvrage accessible sur Algol 68. Depuis, d'autres ouvrages ont été réalisés mais leur diffusion a été assez restreinte. Citons "Algol 68R users guide" [WOODWARD] publié en Angleterre par S. Bond et P. Woodward, du RRE, qui est une présentation claire et condensée de l'essentiel du langage pour des lecteurs connaissant Algol 60. Un ouvrage pédagogique a également été réalisé par Peck [PECK] et le groupe Algol de l'AFCET a écrit un manuel complet du langage destiné aux utilisateurs [AFCET-2]. Ce dernier document, dont nous aurions souhaité un équivalent au moment de la parution du rapport de définition, a vu sa parution officielle retardée par l'annonce de révisions du langage Algol 68. En effet, le groupe de travail WG2.1 a continué à se réunir pour examiner les propositions de changements du langage issues de l'expérience théorique, ou pratique, des premiers utilisateurs et écrivains de compilateurs.

Le rapport de définition du langage modifié est paru en 1974. Les changements sont de plusieurs ordres. Certains concernent seulement la forme externe du langage, d'autres touchent à des aspects plus fondamentaux comme le traitement des modes et des opérateurs. Les différences entre le nouveau langage et l'ancien sont décrites par P. Uzgalis [UZGALIS]. La définition de la syntaxe a été également profondément remaniée et il a été fait un usage plus systématique des possibilités des W-grammaires. Les mécanismes d'identification sont maintenant exprimés dans la syntaxe et l'introduction de prédicats permet souvent une expression plus naturelle des propriétés. Il n'en reste pas moins que, de l'avis même des auteurs, sa lecture peut être difficile pour un lecteur non-initié. Nous déconseillons fortement la lecture du rapport au lecteur qui n'est pas aidé dans cette tâche et qui n'a pas déjà une certaine connaissance du langage. Si ces dernières conditions sont remplies, et moyennant un patient travail d'assimilation il est possible d'en tirer profit.

L'absence de compilateur aussi bien que le manque d'ouvrages lisibles n'ont certes pas facilité la diffusion du langage. Le premier compilateur a été réalisé au Royal Radar Establishment en Angleterre pour les calculateurs de la série ICL 1900, ce qui explique le succès d'Algol 68 dans ce pays. Le RRE est un centre qui n'emploie qu'un seul langage de programmation pour

ses applications, aussi bien de gestion, analyse numérique ou recherche opérationnelle, que pour l'écriture de logiciel. Le langage alors utilisé était un dérivé d'Algol 60 et il avait été décidé de lui trouver un successeur. Algol 68 a été choisi. En fait, quelques aménagements lui ont été apportés, ce qui l'a fait nommer Algol 68-R. Certains de ces changements sont des restrictions dues à des contraintes de compilation en un seul passage et n'altèrent pas la puissance du langage. D'autres ont été apportés délibérement. Ils ont d'ailleurs été adoptés pour la plupart dans la version révisée du langage. Le compilateur est écrit dans le langage de type Algol 60 précédemment cité. Il est accompagné d'un système d'entrées-sorties complet et permet une compilation modulaire des programmes. Une version opérationnelle du compilateur a été terminée en 1970 [CUR-68R]. Elle est utilisée dans ce centre et dans des universités anglaises. Une deuxième version a été écrite en Algol 68-R, en utilisant le premier compilateur, puis autocompilée. Ses performances sont très satisfaisantes.

Citons également l'implantation réalisée par une équipe du MBLE pour les calculateurs Philips Electrologica X8 [BRANQU-2]. En effet, il s'agit de la première implantation du langage Algol 68 sans modification, tel qu'il a été défini dans sa première version. Elle est accompagnée d'une documentation remarquable et d'une étude complète des problèmes posés par la compilation d'Algol 68 [BRANQU-3]. Vu le type d'ordinateur sur lequel elle est réalisée, cette implantation n'a pas connu une grande diffusion.

On peut s'étonner qu'en 1975 il n'existe pas de version opérationnelle du langage Algol 68 sans restriction sur les principaux ordinateurs. Les compilateurs réalisés à l'heure actuelle sont souvent incomplets (gestion de fichiers limitée, absence de ramasse-miettes, tableaux à une seule dimension, pas de tableaux flexibles ...) ou imposent des contraintes dues au choix d'une méthode particulière d'implantation. Un groupe international, dont nous sommes membres, le III (International Implementers Interchange), assure la diffusion des informations touchant à Algol 68, en particulier son utilisation et son implantation, et organise une rencontre annuelle [III-74] [III-75]. Le groupe français de l'AFCET a également permis des échanges fructueux avec les personnes touchées par Algol 68 à divers titres : enseignement, utilisation, écriture de compilateurs. Nous avons participé à ses activités dans le cadre de l'écriture du manuel complet du langage, ou de réunions plus importantes [AFCET-3]. Il a aussi parrainé la traduction française du rapport de définition du langage [AFCET-1].

Notre travail sur un compilateur Algol 68 pour un calculateur IBM 360 a débuté en 1970 par des études préliminaires sur la syntaxe [68GRE-2]. La conception d'un prototype a commencé en 1972 et son écriture s'est terminée fin 1974. Elle a concerné cinq personnes : Messieurs Cunin, Delaunay, Simonet, Voiron et Willis [68GRE-3]. Le séjour de Ian Currie à Grenoble en 1973 a été l'occasion d'échanges fructueux. B. Willis a quitté le groupe en 1973 après avoir soutenu une thèse [WILLIS]. Dans le cadre d'un projet IRIA-SFER [IRIASFER], comportant l'écriture d'un ensemble de compilateurs en collaboration avec l'Université de Rennes, M. Delaunay, après avoir quitté le groupe, dirige la réécriture du compilateur pour un calculateur CII-IRIS 80. B. Bretagnolle travaille actuellement avec P.Y. Cunin et J. Voiron à la transformation du prototype en une version opérationnelle pour le calculateur IBM 360.

Notre premier objectif dans ce travail a été la conception et la réalisation d'un prototype, ainsi que la description de la fonction de traduction, dont notre compilateur est une application particulière. La conversion du prototype en une version opérationnelle est un objectif ultérieur qui demande la mise en oeuvre de moyens spécifiques. Ce travail est déjà avancé, puisque l'exemple donné en annexe a été compilé sur cette version. La décision de réaliser un prototype du compilateur nous a conduits à l'utilisation d'outils de programmation évolués. Certains choix furent heureux, d'autres moins. L'utilisation du langage CDL [KOSTER-2] et d'un générateur d'analyseurs LL(1) [GRIFF-1] s'est révélée satisfaisante. Par contre, le choix du langage PL/1 s'est avéré déraisonnable pour un projet de cette envergure. En effet, la taille des programmes produits aussi bien que leur temps d'exécution se sont montrés prohibitifs et nous avons continué avec le langage PL360, non sans quelques problèmes d'interface avec le langage PL/1.

La version opérationnelle du compilateur est basée sur le langage PL360. L'autocompilation de CDL et la production d'une sortie PL360 pour le générateur d'analyseurs LL(1) [BRE-DEL] ont montré l'intérêt d'employer des outils de haut-niveau qui sont facilement adaptables. La réécriture des programmes écrits en PL/1 ne pose pas de problème particulier, mais le travail ne se limite pas à une simple conversion du prototype. En effet, certaines solutions qui peuvent convenir à une version expérimentale doivent être adaptées ou remplacées pour la version opérationnelle. Il faut également

prévoir l'écriture effective de certaines parties dont l'étude seulement a été faite (entrées-sorties [68GRE-1], calcul parallèle [68GRE-8], tableaux flexibles et ramasse-miettes).

Le langage Algol 68 dont nous parlons dans cette thèse est le langage défini par le rapport révisé paru en 1974 [VANWIJ-2]. Dans cette thèse le rapport de définition sera appelé le Rapport. Nous avons utilisé certains termes techniques spécifiques à Algol 68 et supposé connus de nombreux éléments du langage. Les termes employés sont en général ceux définis par le groupe Algol de l'AFCET, et pour leur définition le lecteur se reportera au manuel écrit par ce groupe [AFCET-2]. Nous conseillons également la lecture de cet ouvrage à ceux qui veulent acquérir une bonne connaissance d'Algol 68. Comme Algol 68 sert d'illustration à notre méthode, nous pensons que cette connaissance du langage est souhaitable pour la lecture de cette thèse.

I - DEFINITION DE LA FONCTION DE TRADUCTION

Notre propos est l'étude des problèmes liés à l'écriture d'un compilateur. L'approche que nous adoptons ici est essentiellement pragmatique et est issue de l'expérience que nous avons acquise au cours de l'écriture d'un compilateur pour le langage Algol 68. Le langage est défini formellement par un rapport [VANWIJ-2] rédigé par le groupe de travail WG 2.1 au sein de l'IFIP.

Cette définition formelle est fondée d'une part sur l'utilisation d'une grammaire à deux niveaux, d'autre part sur une méthode de définition de la sémantique s'apparentant aux méthodes interprétatives,malgré l'utilisation d'un langage pseudo-naturel (sous forme d'anglais formalisé). Dans la version précitée du Rapport, que nous avons utilisée, l'emploi de prédicats dans l'écriture des règles grammaticales a permis, entre autres, de prendre en compte certains aspects contextuels de la syntaxe tels que les problèmes d'identification et le traitement des modes.

Cette définition présente un grand intérêt pour l'écrivain de compilateurs : tous les aspects du langage, même le système d'entrées-sorties (et il faut souligner le mérite du Rapport), sont décrits. Pour ce dernier, l'interface avec le système d'exploitation est parfaitement localisée et plusieurs niveaux de "non-définition" ("undefined") sont introduits dans le Rapport. Ceci est gênant pour l'utilisateur et la portabilité des programmes qu'il écrit (les solutions adoptées pour lever les "non-définitions" au niveau des différents compilateurs, n'étant vraisemblablement pas toutes identiques). On perçoit, à partir de cet exemple, les limites d'une définition formelle de langage. Si pour le théoricien la solution proposée par le Rapport est acceptable, elle ne l'est que partiellement pour l'écrivain de compilateurs. Il paraît difficile à ce niveau, de proposer une définition universelle, absolument indépendante de toute machine ou classe de machine, qui reste réaliste et évite toute imprécision.

Le Rapport ne précise pas, d'autre part, de représentations particulières, pour certaines notions (par exemple, la notion d'environnement d'élaboration, suffisante au niveau de la définition formelle, ne l'est plus pour une implantation particulière).

Il est alors du devoir de tout traducteur de signaler à l'utilisateur les endroits où il a pris une décision non-indiquée dans le Rapport.

La définition, cependant, peut servir de guide à une implantation ; mais il reste difficile d'envisager un passage automatique de l'une à l'autre. Même si ceci était techniquement réalisable, le résultat obtenu serait un compilateur assez inefficace : un exemple extrême est fourni par la définition de certains opérateurs standards définis, pour les besoins du théoricien, en fonction d'autres (l'opérateur + est défini en fonction de l'opérateur -).

La bonne structuration de la définition permet cependant une transposition manuelle assez systématique, comme nous le verrons par la suite, ce qui à notre avis est important, car l'écrivain de compilateur peut avoir une présomption quant à la correction de certaines parties de son compilateur.

D'une manière plus générale, trois groupes de personnes sont intéressés par un langage de programmation : les utilisateurs, les théoriciens et les écrivains de compilateurs. Leurs intérêts ne sont pas identiques. La même définition formelle d'un langage ne peut que très difficilement satisfaire ces trois groupes [GRIFF-3]. Lorho rappelle les principales techniques de définition utilisées et montre leurs avantages et leurs faiblesses pour chacun [LORHO]. On peut affirmer que le Rapport de définition du langage Algol 68 n'échappe pas à la règle et ne peut satisfaire tout le monde.

Nous nous sommes délibérement placés, pour ce travail, dans le groupe des écrivains de compilateurs.

Intérêt d'une description -

Nous avons dans ce travail deux objectifs. Nous voulons :

- écrire un compilateur en essayant de nous dégager des recettes techniques et d'une inspiration plus ou moins intuitive et approximative de la définition formelle du langage, pour nous intéresser plutôt à l'aspect méthodologique,

- décrire le compilateur avec suffisamment de précision et par suite essayer de définir une méthode de description de compilateurs, qui pourrait éventuellement aboutir à une formalisation plus ambitieuse.

Pour réaliser ces divers objectifs nous avons centré notre travail autour de la description du compilateur, que nous proposons dans la suite. Cette description permet de définir complètement le compilateur, elle propose des éléments nécessaires à une définition formelle de la fonction de traduction

et peut être considérée comme un premier pas vers une "validation" du compilateur. En effet, si sa structure est analogue à celle de la définition du langage, il serait souhaitable de pouvoir montrer l'équivalence entre ces deux définitions. D'autre part, si la production du compilateur à partir de cette description peut être rendue systématique, pour peu que l'on puisse montrer la correction de la transformation utilisée, le compilateur se trouverait validé.

La description permet de dégager toutes les fonctions du compilateur, et sert de support à la présentation des divers algorithmes et solutions techniques adoptés dans son écriture. On peut ainsi éviter la description fastidieuse de l'ensemble des "recettes" utilisées (qui restent en général très classiques et sont plus ou moins imposées par la technologie du matériel sur lequel est réalisé le compilateur). On peut par contre montrer plus simplement l'architecture de l'ensemble du système et donner ainsi les éléments d'une méthodologie permettant d'en maîtriser la complexité. Ce dernier point nous apparaît essentiel dans la réussite d'un "bon compilateur".

Méthode de description

Nous nous sommes efforcés d'adopter certains aspects méthodologiques de conception, mis en valeur par les techniques de programmation structurée et d'analyse descendante [DIJKSTRA]. Par ailleurs nous avons gardé la structure de la définition formelle du langage qui sert ainsi fréquemment de base de départ pour la description.

Le niveau de description adopté permet souvent de déduire plusieurs solutions techniques pour l'écriture effective du compilateur. Nous avons séparé dans la suite de la présentation deux aspects complémentaires pour la conception et l'écriture d'un compilateur ; nous présentons successivement :

- la description de la fonction de traduction,
- une représentation possible de la fonction de traduction.

On doit toutefois noter que nous avons parfois délibérement abandonné nos soucis de formalisation lorsque nous obtenions une description trop éloignée des spécifications réelles du compilateur, ou trop complexe par rapport à la réalité. Ainsi nous avons abandonné l'utilisation d'attributs

sémantiques et l'approche déclarative très intéressante qu'ils permettent, chaque fois qu'une approche algorithmique évitait la multiplication de notations et la lourdeur d'un formalisme utilisant des attributs de synchronisation. De même nous précisons de manière informelle certains traitements classiques (identification des indicateurs, par exemple) dont la mise en forme alourdirait la description. Malgré ces restrictions il nous semble que la description proposée peut servir de base à une définition formelle de langage plus adaptée aux soucis de l'écrivain de compilateurs. Il est nécessaire pour juger son intérêt d'analyser les réactions éventuelles et prévisibles des principaux groupes d'intéressés.

Les écrivains de compilateurs peuvent être déroutés par la description et sceptiques quant à l'efficacité du compilateur obtenu. La séparation au niveau de la description de certaines fonctions, l'introduction d'arbres sont autant de points qui peuvent laisser prévoir un compilateur inefficace. Nous montrons cependant, dans la seconde partie de la présentation, que le choix de techniques particulières, le regroupement de certains traitements permettent d'obtenir un résultat tout à fait acceptable : le compilateur que nous avons réalisé pour un calculateur IBM 360 pouvant en fournir une preuve. Notre expérience de réécriture du prototype (pour le même calculateur) peut fournir également une preuve sur la bonne structure de son architecture. Il est à noter également que l'organisation des diverses tâches dans le compilateur, induite par la description, est peu différente de celle proposée par Mailloux [MAILLOUX]. Afin de rassurer l'écrivain de compilateurs, nous avons évité de trop développer certains aspects formels, et inséré dans la description quelques considérations pratiques susceptibles de guider une réalisation effective.

Les théoriciens trouveront vraisemblablement sans intérêt une telle description qui reste encore très incomplète ; nous suggérons néanmoins quelques éléments à partir desquels une voie pourrait être ouverte, permettant peut-être, de prouver la validité des compilateurs.

Les utilisateurs enfin, pourront peut être évaluer avec plus de précision le coût de certains concepts introduits dans le langage et se faire une "opinion" que nous espérons honnête pour peu que la lecture de cette description ne soit pas plus fastidieuse que celle du rapport de définition du langage lui-même. Notons qu'il ne s'agit pas, malgré tout, d'une définition "accessible" à tout utilisateur.

Après cette brève justification de l'intérêt d'une telle description fonctionnelle nous allons en préciser le principe.

1 - DESCRIPTION DE LA FONCTION DE TRADUCTION -

On peut distinguer, lorsque l'on aborde les problèmes de langages, les trois notions suivantes :

- l'expression ou écriture d'un programme dans le langage
- le sens ou valeur sémantique d'un programme écrit dans ce langage
- la correspondance qui à tout programme du langage associe un sens.

Cette correspondance est effectuée en général en deux étapes. Dans notre cas (approche traductive) le programme écrit dans le langage (programme source) est traduit en un programme équivalent écrit dans un langage machine (programme objet) à partir duquel est faite l'élaboration qui permet de donner un sens au programme initial.

La correspondance entre programme source et programme objet est établie par une fonction de traduction (ou traduction). C'est cette fonction que nous allons étudier et décrire,et qui définit le compilateur.

1.1 - Traitements sémantiques et langage noyau -

Il n'est pas très commode de définir le sens d'un programme à partir de son écriture linéaire. Par exemple, si l'on considère le programme :

```
début
    ent a, b ; lire(a) ;
        b := a ;
        écrire(b)
fin
```

il est nécessaire pour élaborer $b := a$ de reconnaître les deux constituants de l'affectation (source et destination), de les élaborer collatéralement, d'affecter enfin le résultat de la source à la destination. Toutes ces informations ne sont pas mises en évidence par l'écriture linéaire. S'il est difficile de définir l'élaboration du programme, il est tout aussi difficile d'en définir la traduction en un programme objet équivalent. Il paraît plus

adapté pour définir le sens d'une phrase d'utiliser une représentation mettant en évidence une structure syntaxique dont les composants sont utilisés comme support de cette définition sémantique.

Nous reprenons une démarche fréquemment utilisée dans les méthodes de définition formelle de la sémantique. Nous allons définir un langage pivot (ou noyau car il ne comporte que des constructions en nombre limité, plus simples que celles du langage que l'on veut compiler). D'après la constatation précédente, ce langage est formé d'arbres dont nous préciserons ultérieurement la forme et la nature exacte ; il sert de support pour le traitement sémantique du programme.

Nous pouvons alors préciser le schéma de traduction :

1°) - le programme source est transformé en un programme du langage noyau

2°) - à partir de cette nouvelle expression du programme, plus adaptée au traitement sémantique, on termine la traduction. Nous pouvons faire plusieurs remarques sur un tel schéma.

Langage noyau et définition formelle du langage -

Nous montrons, à partir d'un exemple, que la définition formelle du langage repose sur le même principe. Si, pour la traduction, nous choisissons un langage noyau dont la forme est semblable à celle utilisée dans la définition formelle, il est relativement facile de définir les traitements sémantiques, pour une partie du compilateur (en adaptant les paragraphes sémantiques de cette définition).

La définition formelle du langage Algol 68, repose, nous l'avons vu, sur une grammaire à deux niveaux. La technique utilisée permet, d'une part, de prendre en compte dans la syntaxe, certains aspects du langage que l'on ne peut pas exprimer à l'aide d'une syntaxe hors-contexte, d'autre part, de fournir une structure commode pour la définition sémantique. Nous allons examiner ce dernier point sur un exemple. L'ensemble du Rapport est organisé en paragraphes (regroupés eux-mêmes en chapitres) qui comportent, en plus des règles de grammaire, une partie ("semantics") définissant la sémantique associée à ces règles.

Considérons la définition de l'affectation (§ 5.2.1. du Rapport)

5.2.1 Assignations
5.2.1.1 Syntax
a) REF to MODE NEST assignation ;
   REF to MODE NEST destination, becomes token, MODE NEST source
b) REF to MODE NEST destination : soft REF to MODE NEST TERTIARY
c) MODE1 NEST source : strong MODE2 NEST unit,
        where MODE1 deflexes to MODE2.

5.2.1.2 Semantics
a) An assignation A is elaborated as follows :
   - let N and W be the yields of the destination and source of A ;
   - W is assigned to N ;
   - the yield of A is N.
b) ...

La structure induite par cette définition, est une structure d'arbre proche de celle qui serait induite par l'utilisation d'une grammaire de définition hors-contexte. Nous allons essayer de caractériser sur cet exemple la forme de l'arbre.

Les symboles non terminaux "destination" et "source" ne sont pas nécessaires à la description syntaxique de l'affectation, ils permettent de "nommer" des sous-arbres de l'arbre syntaxique. La sémantique est définie à partir de ces sous-arbres. On considère, en effet, dans la définition sémantique, le "yield of the source" et le "yield of the destination" qui sont les résultats de l'élaboration des sous-arbres induits respectivement par "unit" et "tertiary". L'élaboration d'un programme procède par l'élaboration "des sous-arbres", selon un processus de réduction postfixée partant des feuilles et aboutissant à la racine de l'arbre associé à ce programme. On peut ainsi déduire de la définition formelle la forme d'un arbre privilégié, utilisé pour la définition de la sémantique et,par suite,pour la traduction.

1.2 - Aspects statiques et dynamiques des traitements sémantiques -

On distingue, d'autre part, deux classes de traitements sémantiques :

- ceux qu'il est possible de faire statiquement, et qui,dans notre cas, sont effectués sur la représentation du programme utilisant le langage noyau,

- ceux qu'il n'est possible de faire que dynamiquement et pour lesquels une génération de code objet est nécessaire.

Les traitements sémantiques statiques, effectués pendant la traduction permettent d'obtenir une meilleure efficacité du programme objet. Nous donnons à titre d'exemple les deux principaux traitements en Algol 68 :

- l'"identification", associant une occurrence d'utilisation à une déclaration,
- les vérifications concernant les modes et la génération d'appels de conversion, appelées "modifications de valeurs" en Algol 68.

On peut noter que si ces traitements ne sont pas exprimables par une grammaire hors-contexte, ils apparaissent, dans la définition formelle du langage Algol 68, au niveau de la syntaxe. On peut, par suite, qualifier cet aspect de la sémantique,de syntaxe contextuelle. Nous ne limitons pas les traitements statiques de la sémantique à ces deux cas ; nous rangeons dans cette catégorie tout traitement sémantique qui peut être fait à la compilation sans changer la signification du programme. Nous pouvons maintenant préciser la forme du langage noyau et le principe de la traduction.

2 - FORME DU LANGAGE NOYAU -

Ce langage est formé d'arbres ou de ramifications au sens défini dans [PAIR]. Ces ramifications pourraient être construites à partir des "marqueurs de phrases" pour une grammaire engendrant le langage, mais un tel choix a deux inconvénients :

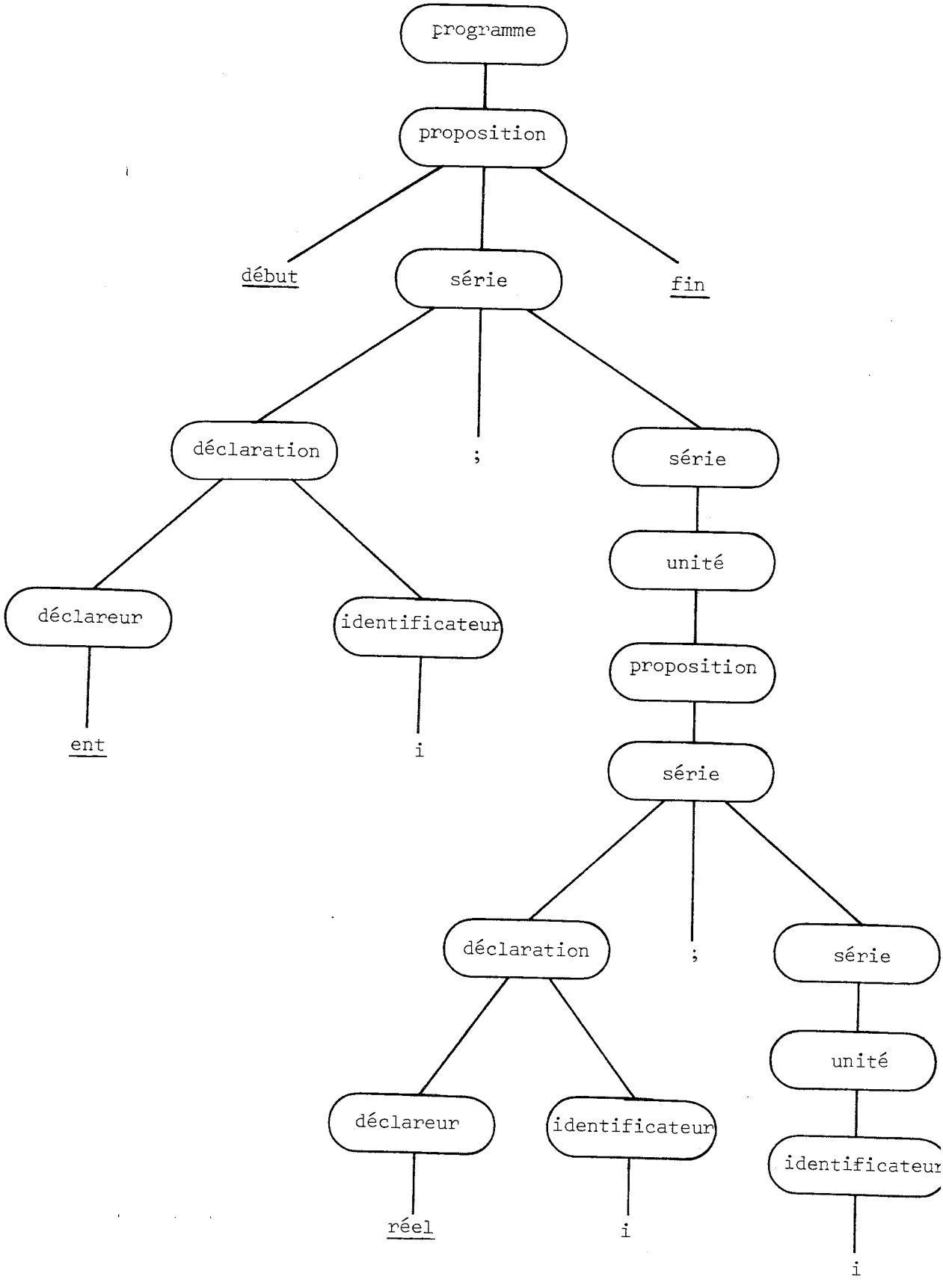
- Ces arbres contiennent des informations sans intérêt pour la "signification du programme". On peut reprendre la démarche de Mac Carthy [MACCARTH] et supprimer ces informations que Chomsky et les linguistes qualifient pour les langues naturelles "de surface", par opposition aux "structures profondes" auxquelles est attachée la signification d'une phrase. On peut ainsi introduire la notion de syntaxe abstraite du langage (les programmes sont représentés dans le langage noyau sous forme d'arbres) par opposition à syntaxe concrète (représentation linéaire).
- La grammaire utilisée peut d'autre part induire un arbre sur lequel les informations utiles au traitement sémantique sont "mal placées" pour que la "signification" d'un sous-arbre ne dépende que de ce sous-arbre et non de

son contexte. La structure de l'arbre est alors mal adaptée au traitement sémantique et à la traduction. Cet arbre n'a donc pas la propriété que nous qualifions par la suite de propriété "de localité de la traduction". C'est pourquoi, nous utilisons, comme langage noyau, des arbres qui ne présentent pas ces inconvénients. Nous appelons arbre abstrait la représentation du programme source à l'aide de ce langage.

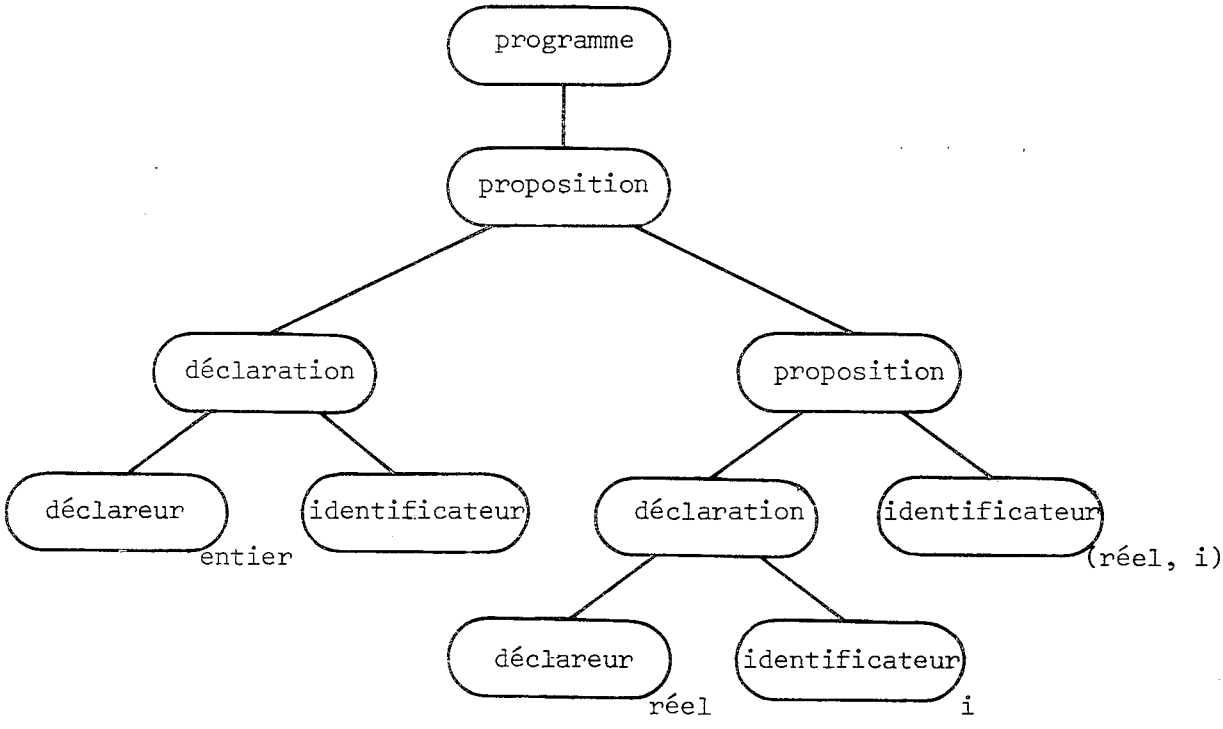
Nous illustrons les deux points précédents en montrant l'arbre syntaxique et l'arbre abstrait d'un programme. Nous utilisons pour cela la grammaire et l'exemple proposés dans le chapitre II.1 au paragraphe 1.1. L'exemple est le suivant :

```
début
    ent i ;
    début
        réel i ;
        i
        .
        .
        .
    fin
fin
```

Le premier point est mis en évidence par la différence de taille entre l'arbre syntaxique et l'arbre abstrait. De nombreux noeuds de l'arbre syntaxique sont sans intérêt pour la "signification" du programme. Ils ont disparu dans l'arbre abstrait, qui ne conserve que les noeuds essentiels. Le principe de localité n'est pas respecté sur l'arbre syntaxique. En effet, le mode de l'identificateur i est attaché à sa déclaration. A l'endroit de son utilisation, le mode de i n'est pas connu. Dans l'arbre abstrait, le mode de l'identificateur est attaché à son utilisation. L'arbre abstrait pour cette raison doit être appelé plus justement, "arbre abstrait décoré".



Arbre syntaxique



Arbre abstrait

Le langage noyau est défini par une syntaxe que nous donnons en annexe (voir annexe 1). Cette syntaxe est qualifiée de syntaxe abstraite par opposition à la syntaxe concrète (forme linéaire du programme source) en utilisant la terminologie proposée par Mac Carthy et reprise par la méthode de Vienne [LUCAS]. La syntaxe abstraite est définie à partir d'une grammaire de ramifications.

3 - SCHEMA DE LA TRADUCTION -

Nous pouvons maintenant préciser entièrement la forme du schéma de traduction et les diverses fonctions de cette traduction, dont on donne la description complète dans la partie suivante.

Nous définissons une première transformation sur le programme source permettant de passer à une première représentation arborescente ; cette transformation peut, en pratique, être réalisée au cours de l'analyse syntaxique. La représentation arborescente, ainsi obtenue, présente, en général, les deux inconvénients, cités précédemment au paragraphe 2. On peut les supprimer en définissant deux nouvelles transformations t1 et t2 sur l'arbre formé à partir des marqueurs de phrases :

t1 permet d'obtenir l'arbre abstrait primitif ; ce nouvel arbre représente la structure profonde du programme.

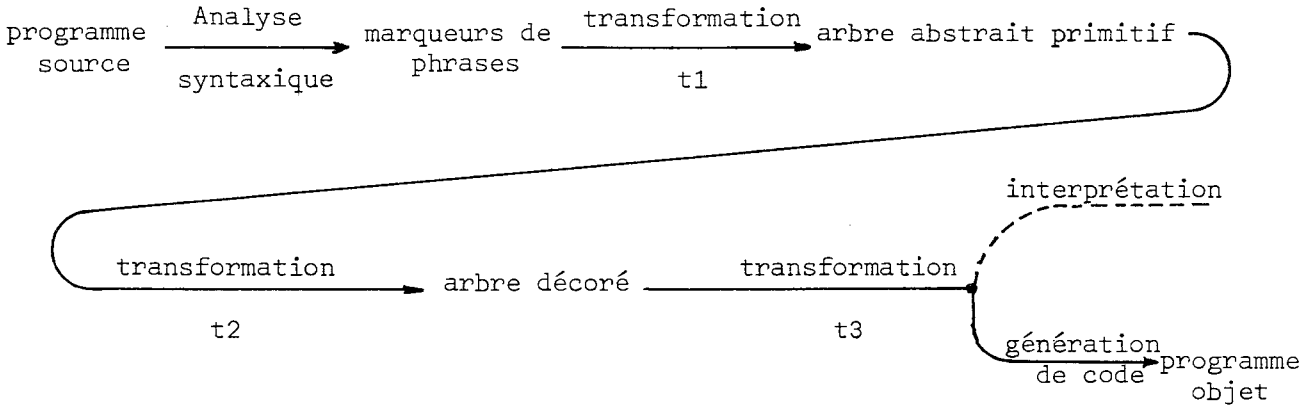
t2 permet d'obtenir l'arbre abstrait décoré pour lequel la propriété de localité est vérifiée.

Dans la pratique la transformation t1 peut être réalisée au cours de l'analyse syntaxique. Pour effectuer la transformation t2 de la même manière, il est nécessaire d'adopter certaines "entorses" vis à vis de la définition formelle du langage. Nous envisageons ce problème dans la troisième partie.

Les décorations, ajoutées sur l'arbre abstrait primitif ne sont en fait que des informations locales, représentables sur cet arbre, sous forme d'attributs au sens de Knuth [KNUTH-2], par exemple. Ces informations ne sont privilégiées, et ainsi rattachées aux noeuds que parce qu'elles servent de paramètres à la génération de code ou à l'interprétation. Dans la pratique certaines décorations peuvent perdre leur caractère privilégié qui les rattache à un arbre,et redevenir des informations locales.

La dernière transformation, t3, définit à partir de l'arbre abstrait décoré le processus de production de la séquence de code objet, dernière étape de la traduction. Au niveau de la description de la fonction de traduction, donnée dans la partie suivante, nous avons préservé la possibilité de remplacer le générateur de code, par un interpréteur dont l'entrée est fournie par une représentation de l'arbre abstrait décoré. C'est une des raisons pour lesquelles la description de la transformation t3 utilise des "modèles" (voir ch. II.4.5).

Toutefois, nous nous intéressons essentiellement à une fonction de traduction utilisant un processus de génération de code. Le schéma général de la traduction peut être résumé comme suit :



Dans la pratique on peut confondre la phase d'analyse syntaxique et tout ou partie des transformations t1, t2, t3. Nous reviendrons sur ce problème dans la troisième partie de cette thèse.

II - DESCRIPTION DE LA FONCTION DE TRADUCTION

Dans cette partie nous donnons une description du processus de compilation du langage Algol 68. Cette description est conçue pour servir de guide à l'écrivain d'un compilateur. Elle ne préjuge pas de solutions techniques particulières. Nous donnons brièvement, dans la troisième partie, (voir ch. III) les solutions adoptées dans le cas de l'implantation que nous avons réalisée sur un calculateur IBM 360. Toutefois, pour faciliter la lecture de la description,nous faisons référence, dans cette partie,à certains éléments plus techniques. Ces indications ne doivent être considérées que comme des solutions possibles au problème traité, d'autres solutions étant envisageables.

La description du processus de compilation est divisée en deux parties, reliées par l'arbre abstrait décoré, image du programme à compiler. La première étape décrit la construction de l'arbre abstrait à partir du programme source, ainsi que sa décoration. La deuxième étape décrit la production de code à partir de l'arbre décoré. Avant d'entrer dans le détail de la description, nous allons préciser les outils utilisés dans la description et montrer son principe à partir d'un exemple.

II.1 - OUTILS DE LA DESCRIPTION -

La construction de l'arbre abstrait primitif ne nécessite pas l'emploi d'un système d'attributs. En effet, on peut en faire une description par récurrence sur l'arbre syntaxique, puisque le résultat sur un sous arbre ne dépend que de ce sous arbre. Il n'en est pas de même pour sa décoration. Dans ce cas, les informations concernées sont réparties dans l'arbre et il faut les transmettre à l'endroit de leur utilisation. C'est le cas pour les processus d'identification des identificateurs et des opérateurs ainsi que pour le traitement des modes. L'utilisation des attributs s'est avérée assez bien adaptée à leur description.

Pour n'employer qu'un seul outil, nous avons également utilisé les attributs pour décrire la construction de l'arbre abstrait primitif. Il suffit dans ce cas d'un seul attribut, qui représente l'arbre abstrait, et qui est synthétisé. Nous allons rappeler la notion classique d'attribut sur une grammaire hors-contexte et introduire les attributs sur un arbre. Ces attributs sont utilisés dans les phases de décoration. Nous donnons ensuite les éléments de notations algorithmiques introduits dans la description. Nous précisons enfin les notations concernant la représentation des listes.

1 - LA NOTION D'ATTRIBUT -

Les attributs ont été introduits par Knuth pour définir la "signification" de programmes [KNUTH-2]. Depuis, la notion d'attribut a donné lieu à plusieurs réalisations dont l'objectif est de permettre l'implantation et la définition de langages [FAN-FOLD] [BO-SOFT] [LORHO].

Les attributs sont des valeurs attachées aux symboles non-terminaux d'une grammaire hors-contexte. Ils sont définis par des règles d'évaluation associées à chaque règle de production de la grammaire. Dans une règle $A_o \rightarrow A_1 \ldots A_i \ldots A_n$, les attributs calculés pour A_o sont dits synthétisés car ils permettent de "remonter" l'information contenue dans les sous-arbres correspondants. Les attributs calculés pour les non-terminaux de droite $A_1 \ldots A_n$ sont dits hérités car ils vont "descendre" l'information sur les sous-arbres correspondants. Nous donnons un exemple de système d'attributs qui permet de résoudre le problème de la relation entre l'utilisation d'un identificateur et sa déclaration, dans un langage de programmation à structure de bloc.

1.1 - Exemple de système d'attributs -

Le langage servant de support à l'exemple est un langage à structure de blocs. Tout en étant simplifié à l'extrême, il permet d'introduire quelques notions utilisées par la suite.

Les valeurs manipulables sont de mode *entier*, *réel* ou *procédure à un paramètre*, mais pour des raisons de simplicité ces dernières ne sont pas déclarées dans le programme. Ce sont seulement des procédures de librairie. Les déclarations apparaissent dans une "proposition", entre les symboles *début* et *fin*, qui délimitent un bloc. Les instructions sont limitées à l'affectation et l'appel de procédure de librairie. L'identification obéit aux règles habituelles de la structure de bloc et c'est ce processus que nous allons exprimer par un système d'attributs sur une grammaire hors-contexte du langage.

Pour chaque proposition, nous construisons un ensemble formé des couples {mode, nom} des identificateurs. Il est représenté par l'attribut synthétisé SBLOC et il est construit à partir des attributs de chaque déclaration. La règle de la structure de bloc est exprimée par un attribut hérité HBLOC qui, à la racine, a pour valeur l'ensemble des déclarations de la librairie. A chaque proposition, il s'enrichit des déclarations propres à cette proposition, représentées par l'attribut SBLOC. L'identification d'un identificateur consiste à rechercher cet identificateur dans l'ensemble de déclarations HBLOC qui a été descendu jusqu'à lui. Nous donnons maintenant la grammaire avec les règles d'évaluation des attributs, sans expliciter la fonction "identifier" ni les règles de construction de l'attribut NOM attaché à un identificateur (il s'agit par exemple d'un numéro lexicographique). L'objectif est d'associer à chaque identificateur utilisé dans le programme, le mode provenant de sa déclaration. Les conventions d'écriture de la grammaire et des règles d'attributs sont celles données dans le chapitre II.3.2 au paragraphe 2.

Grammaire et règles de définition d'attributs

programme

→ proposition

| HBLOC•proposition ← déclarations standard(lire, écrire, sin, ...)

proposition

→ *début*, série, *fin*

| HBLOC•série ← SBLOC•série + HBLOC•proposition

série$_1$

→ déclaration, ';', série$_2$

| SBLOC•série$_1$ ← SBLOC•déclaration + SBLOC•série$_2$

| HBLOC•série$_2$ ← HBLOC•série$_1$

→ unité, ';', série$_2$

| SBLOC•série$_1$ ← SBLOC•série$_2$

| HBLOC•unité ← HBLOC•série$_1$

| HBLOC•série$_2$ ← HBLOC•série$_1$

→ unité

| SBLOC•série$_1$ ← ∅

| HBLOC•unité ← HBLOC•série$_1$

déclaration
→ déclareur, identificateur
| SBLOC•déclaration ← {MODE•déclareur, NOM•identificateur}

déclareur
→ *ent*
| MODE•déclareur ← entier

→ *réel*
| MODE•déclareur ← réel

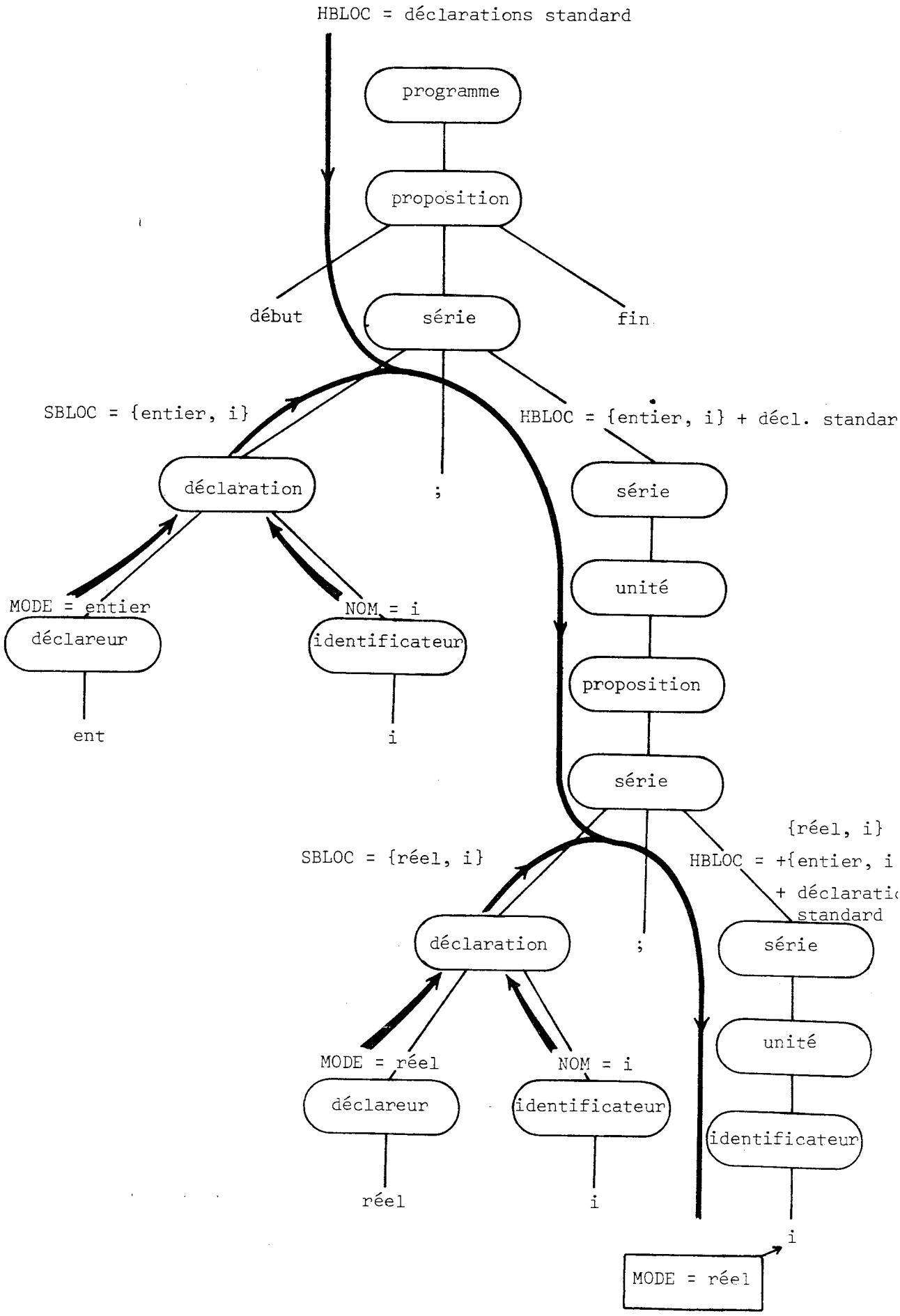
unité
→ identificateur
| MODE•identificateur ← identifier (NOM•identificateur, HBLOC•unité$_1$)

→ proposition
| HBLOC•proposition ← HBLOC•unité

L'attribut MODE étant associé à l'identificateur, le processus d'identification est terminé pour cet identificateur. L'information MODE, qui était attachée à sa déclaration, est maintenant devenue locale, pour son utilisation.

Nous illustrons le flot des attributs sur l'arbre syntaxique correspondant au programme suivant :

```
début
    ent i ;
    ...
    début
        réel i ;
        ...
        i
    fin
fin
```

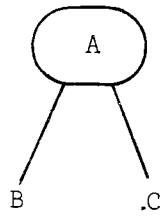

HBLOC = déclarations standard
programme
proposition
début
série
fin
SBLOC = {entier, i}
HBLOC = {entier, i} + décl. standar
déclaration
;
série
unité
MODE = entier
déclareur
NOM = i
identificateur
proposition
ent
i
série
{réel, i}
SBLOC = {réel, i}
HBLOC = +{entier, i
+ déclarati
standard
déclaration
;
série
unité
MODE = réel
déclareur
NOM = i
identificateur
identificateur
réel
i
i
MODE = réel

Nous nous proposons maintenant de décrire un système d'attributs sur un arbre sans passer par l'intermédiaire d'une grammaire.

1.2 - Attributs sur un arbre -

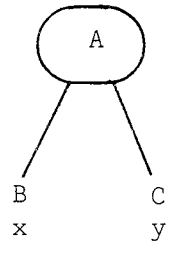
Dans la construction de l'arbre abstrait nous avons utilisé une étape intermédiaire qui est l'arbre primitif. L'arbre primitif d'un programme peut être considéré comme l'arbre syntaxique de ce programme relativement à une grammaire. Or une telle grammaire est de peu d'intérêt pour nous. Nous décrivons donc directement le système d'attributs sur l'arbre primitif.

Nous appelons arbre une arborescence orientée de gauche à droite dont les noeuds sont étiquetés [PAIR]. Plutôt que le formalisme algébrique développé par PAIR, nous avons choisi pour représenter les arbres une forme imagée qui lui est équivalente. Ainsi l'arbre A*(B+C) est représenté par le dessin

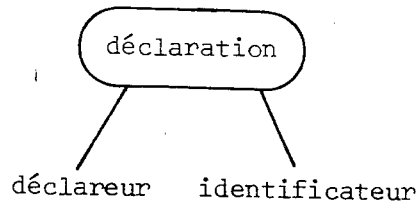


où B et C peuvent être des feuilles, ou d'autres arbres.A cet arbre correspond la règle de grammaire A → B,C. Les attributs sur cette règle sont généralement notés a•A, a•B, a•C. Sur l'arbre, nous pouvons noter de la même manière les attributs attachés au sommet A, mais B et C n'ont pas d'existence en tant que sommets de l'arbre. Comme l'arborescence est orientée, nous pouvons numéroter les fils et noter a(1) les attributs attachés à A et a(2) ceux de B. Or, la grammaire qui sert à engendrer l'arbre abstrait a une forme particulière : le premier membre d'une règle détermine la longueur du second membre. N'utilisant pas cette grammaire, nous avons introduit des sélecteurs pour désigner les fils de chaque noeud, ce qui est possible, car le nombre de fils d'un noeud est connu et ne dépend que de ce noeud.

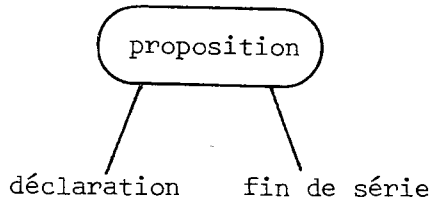
Dans l'exemple ci-contre, le sélecteur x désigne le fils B et y le fils C. Les attributs sont notés a•x et a•y.



Le système d'attributs précédent qui nous a servi d'exemple peut alors être décrit sur un arbre abstrait. Nous donnons quelques règles d'un tel système.

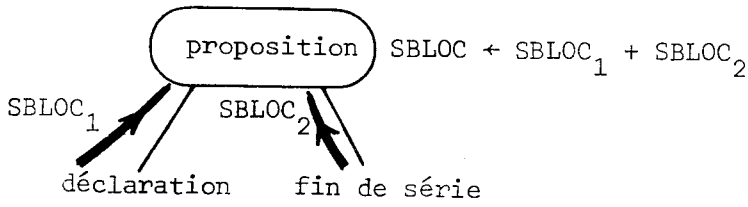


SBLOC déclaration ←
{MODE•déclareur, NOM•identificateu

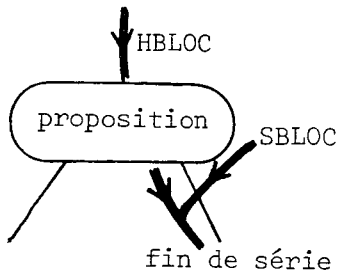


SBLOC•proposition ←
SBLOC•déclaration+SBLOC•findeséri
HBLOC•findesérie ←
SBLOC•proposition+HBLOC•propositi

C'est cette dernière règle qui exprime ici la structure de bloc. Les fils du noeud (proposition) , qui sont des déclarations, servent à constituer l'attribut SBLOC de ce noeud.



L'ensemble de déclarations héritées sur les unités constituant la proposition est formé des déclarations du bloc englobant, représentées par l'attribut HBLOC, et de celles du bloc, représentées par l'attribut SBLOC.



2 - LANGAGE ALGORITHMIQUE UTILISE -

Ce langage algorithmique a pour but d'exprimer certaines parties de la description en termes d'objets et d'actions. Il peut être considéré comme un langage de spécifications, et à ce titre les concepts introduits sont minimaux. Nous utilisons dans ce paragraphe, pour illustrer certaines de ses constructions, des objets appelés des descripteurs. Ces objets décrivent, pendant la compilation, des valeurs Algol 68 intervenant pendant l'évaluation du programme. Nous les introduisons intuitivement à partir d'un exemple au chapitre II.2, et les définissons au chapitre II.4.2 paragraphe 2.2.3.

2.1 - Les objets -

Dans la description nous utilisons plusieurs types d'objets, différents des déclarations attachées aux noeuds, et des attributs définis précédemment. D'un point de vue algorithmique, chaque objet est caractérisé par un identificateur et un type. Le type est précisé, comme en Algol 68, par un indicateur de mode permettant de caractériser l'objet sans préjuger pour celui-ci d'une représentation possible.

Exemple : Lorsque nous voulons déclarer localement un descripteur nous utilisons la déclaration :

descr δ

descr est l'indicateur précisant le type de l'objet déclaré.

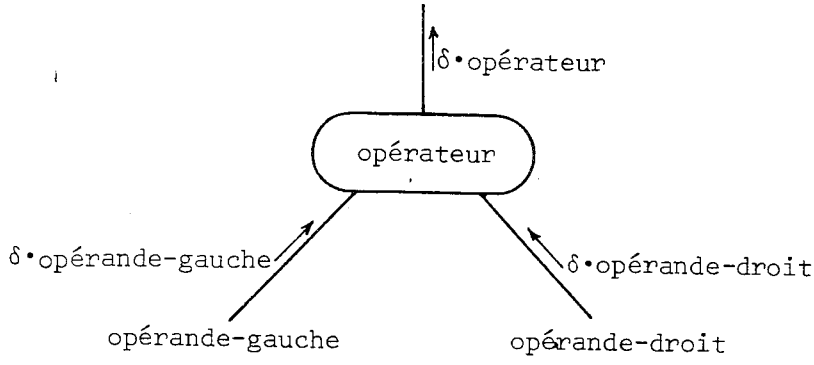
δ est le nom externe de l'objet déclaré.

Les objets sont utilisés de deux manières dans cette description :

- soit localement à une action, auquel cas il leur correspond effectivement dans la réalisation pratique du compilateur une réservation locale d'espace. Ils sont alors déclarés dans cette action.

- soit comme information calculée. Par analogie avec les attributs nous qualifions de synthétisée une information calculée sur un noeud de l'arbre, d'héritée une information calculée pour les fils de ce noeud. Ces objets sont déclarés dans la description sur les schémas des noeuds comme les attributs.

Exemple : Considérons le noeud (opérateur) et les objets descripteurs de valeur associés à ce noeud pendant la génération. Ces descripteurs sont déclarés par :



Groupement d'objets

Il est parfois nécessaire de grouper sous forme d'ensembles ordonnés finis, des objets de même nature. On utilise à cette fin l'indicateur *ensemble* suivi du type des objets de l'ensemble.

Exemple : *ensemble descr* A. A est un ensemble de descripteurs (objets de type *descr*). Un tel ensemble est utilisé dans la description du noeud (index) pour représenter les valeurs des indices. L'ensemble étant ordonné on le construit à l'aide d'une opération de concaténation (union ordonnée) notée +. Au cours de la description, on est amené à définir des opérations sur ces objets.

2.2 - Outils de contrôle utilisés dans la description -

La description étant de nature algorithmique, nous avons introduit divers types de contrôle sur les actions :

- Contrôle collatéral -

Un ensemble de n actions, dont l'évaluation est collatérale, est noté :

collatéral
có n actions co
fincollatéral

- Contrôle séquentiel -

Un ensemble de n actions, dont l'évaluation est séquentielle, est noté :

séquence
    co n actions co
finséquence

- Choix entre plusieurs actions -

* Le choix à deux alternances est noté d'une manière analogue à celui qui est utilisé en Algol 68 :

si condition
  alors
    co *actions* co
  sinon
    co *actions* co
finsi

* Le choix à plus de deux alternances est utilisé dans le cas suivant : Considérons un ensemble E, dans lequel sont définies n parties E_i de fonction caractéristique, $predicat_i$, et la partie $A = E - \cup_i Ei$. On associe à chaque Ei et à A un ensemble d'actions. Pour tout élément x de E on décide d'effectuer l'ensemble d'actions associé :

- à Ei si x appartient à une partie Ei
- à A si x n'appartient à aucune des parties Ei.

cas x
  dans
    prédicat1 :
      co *ensemble d'actions* co,
    prédicat 2 :
      co *ensemble d'actions* co,
      .
      .
      .
  sinon
      co *ensemble d'actions executé par défaut* co
fincas

- Contrôle de répétition -

Un ensemble d'actions peut être élaboré de manière répétitive. L'énumération des éléments d'un autre ensemble E contrôle la répétition. (L'élément courant sert d'élément de contrôle). L'ensemble E fini peut être supposé ordonné mais l'énumération est collatérale.

Pour tout type élément$_i$ ∈ E
faire
A(i, élément$_i$)
co A est l'ensemble d'actions dépendant de i et de élément exécuté de manière répétitive co
finfaire

où *type* est le type des éléments de l'ensemble E et élément$_i$ est le $i^{ème}$ élément de E.
élément$_i$ et i sont disponibles dans la boucle.

2.3 - Structure de la description -

La génération de code est construite à partir d'une hiérarchie d'actions de diverse nature. La description reflète cette structure.

On définit plusieurs types d'actions. Par exemple :

- les actions de compilation (consultation de tables, etc...) notées fonction
- les actions de production de code objet notées modèles
- etc....

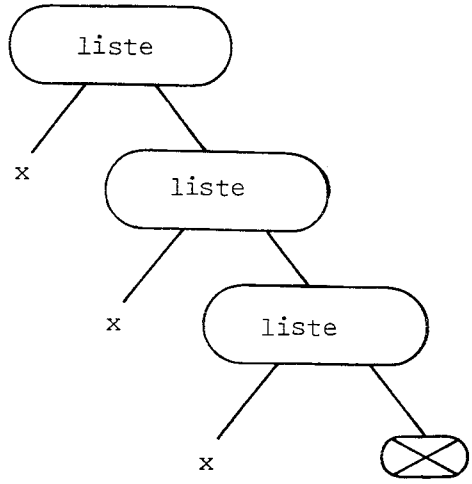
Chacune de ces actions s'écrit suivant le schéma :

action = (co liste des paramètres précédés de leur indicateur de type co)
(co liste des résultats précédés de leur indicateur de type co)
co ensemble d'actions plus élémentaires co
finaction

Remarque : si la liste des résultats ne contient qu'un élément on omet les parenthèses.

3 - REPRESENTATION DES LISTES ET TRAITEMENTS ASSOCIES -

Précisons, enfin, certaines notations propres à la représentation des arbres dans cette description. La notation x-liste représente une liste binaire dont les éléments sont de type x. On peut représenter une telle liste de la manière suivante :

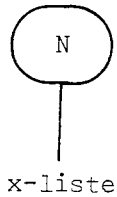


Cette forme est souvent utilisée lorsque le traitement sur la liste est exprimé de manière algorithmique. C'est le cas pour la description de la génération de code, où l'on a besoin d'expliciter la structure de liste sous forme d'une liste binaire.

Dans l'approche déclarative qui est faite avec l'utilisation des attributs, on ne s'intéresse pas à la structure de liste. Nous considérons simplement l'ensemble des éléments de la liste. Nous introduisons ici quelques notations que nous avons utilisées dans la description par attributs de la construction de l'arbre abstrait. Ces notations permettent d'abréger le nombre de règles utilisées pour le calcul, en n'explicitant pas le traitement au niveau de chaque noeud (liste) .

$\{f(x) \mid x \in x\text{-liste}\}$ représente l'image de l'ensemble des éléments de x -liste par la fonction f.

Exemple :

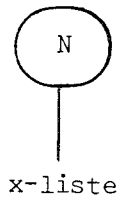


$A \cdot N \leftarrow \{a \cdot x \mid x \in x\text{-liste}\}$

L'ensemble A, attribut du noeud N est formé des attributs a attachés aux composants de la liste.

$(\mathcal{R}(x))^*$ où $\mathcal{R}$ est une règle de calcul d'attributs faisant intervenir les éléments x de la liste x-liste,
est équivalent à l'ensemble de règles
$\{\mathcal{R}(x)$, quel que soit $x \in$ x-liste$\}$

Exemple :

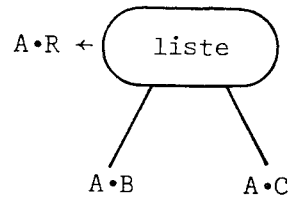


$(a \bullet x \leftarrow f(A \bullet N))^*$
est équivalent à
$\{a \bullet x \leftarrow f(A \bullet N)$, quel que soit $x \in$ x-liste$\}$

Les listes binaires sont construites comme des ensembles ordonnés.

Plutôt que d'écrire :

$R \rightarrow B, C$



où l'arbre A.C est lui-même une liste binaire, nous écrivons :

$$A \bullet R \leftarrow A \bullet B \oplus A \bullet C$$

Dans le traitement qui est fait sur l'arbre abstrait, nous utilisons l'une ou l'autre représentation selon le type de traitement effectué.

4 - CONCLUSION -

Rappelons les outils utilisés au cours des trois étapes de la traduction :

- t_1 - production de l'arbre abstrait primitif par un système d'attributs sur une grammaire hors-contexte.
- t_2 - décoration de l'arbre abstrait par un système d'attributs sur l'arbre abstrait.
- t_3 - production du texte objet par une méthode qui conserve la localité de la traduction. Elle est décrite au chapitre II.4.

Nous allons illustrer dans le chapitre suivant, la méthode de description en suivant sur un exemple les étapes de cette description.

II.2 - INTRODUCTION A PARTIR D'UN EXEMPLE -

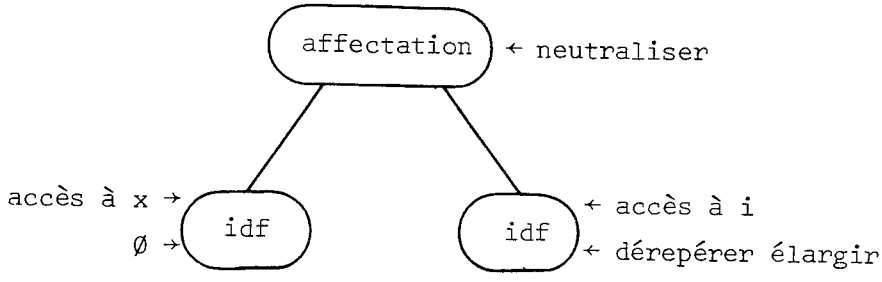
Considérons le programme Algol 68 que nous avons utilisé dans le chapitre I au paragraphe 2

```
début
    réel x ; ent i ;
    ...
    x := i ;
    ...
fin
```

Nous nous intéressons plus particulièrement à l'affectation $x := i$, dans le contexte des déclarations des variables réelle x et entière i. Nous allons suivre son évolution au cours des deux étapes de la description. La première étape est celle de la construction de l'arbre abstrait décoré.

1 - CONSTRUCTION DE L'ARBRE ABSTRAIT DECORE -

L'objectif de cette étape est de construire pour l'exemple donné de l'affectatic l'arbre suivant (qui va servir ensuite à la génération de code) :



Cet arbre est une partie de l'arbre complet construit pour le programme. Il contient toute l'information nécessaire à la génération de code pour l'affectation. Le noeud (idf) qui correspond à un identificateur possède deux décorations : l'accès aux caractéristiques de l'identificateur (son mode, son déplacement dans la zone des variables ...) et la liste des modifications qui lui est attachée.

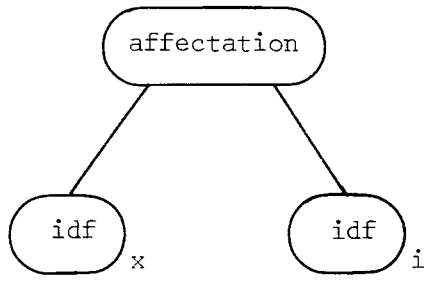
Les modifications en Algol 68 correspondent à la notion classique de conversion. Ainsi l'identificateur *i* doit être *dérepéré* puis *élargi*. Dérepérer *i* signifie que l'on s'intéresse au contenu de la variable et non à la variable elle-même. En d'autres termes, on veut accéder à la valeur contenue à l'adresse spécifiée par *i*. Cette valeur est de mode *ent*. Or la variable *x* demande une valeur de mode *réel*. La valeur entière doit donc être convertie en une valeur réelle. C'est la signification de la modification *élargir* qui suit la modification *dérepérer*. Par contre aucune modification n'est associée à l'identificateur *x*, car sa valeur est considérée telle quelle.

Le noeud *affectation* possède une décoration qui est une liste de modifications. En Algol 68, une affectation a une valeur, qui est celle de sa partie gauche. Cette valeur peut être utilisée dans une autre construction et peut donc être soumise à des conversions. Dans le cas présent sa valeur (un *repère de réel*), n'est pas utilisée. On dit qu'elle est neutralisée, ce qui correspond à la libération des ressources associées à cette valeur (registre ou cellules mémoire utilisés pour représenter cette valeur à l'exécution). La notion de modification peut paraître complexe. En fait, elle permet de structurer le problème des conversions. L'utilisateur n'a pas à s'en soucier dans la plupart des cas, car elle correspond à une utilisation naturelle des valeurs dans un programme. Par contre l'écrivain de compilateurs y trouve un schéma cohérent pour l'implantation des conversions et la vérification statique du bon emploi des valeurs.

La construction de l'arbre abstrait décoré est elle-même décrite en plusieurs étapes : tout d'abord la construction d'un arbre abstrait primitif, puis deux phases de décoration.

1.1 - Construction de l'arbre abstrait primitif -

L'arbre abstrait primitif ne contient que peu de décorations : celles qu'il est commode de mettre en place lors de sa construction. Il s'agit d'informations synthétisées et en général locales à la construction considérée. Dans le cas de l'instruction d'affectation considérée, on construit l'arbre suivant :

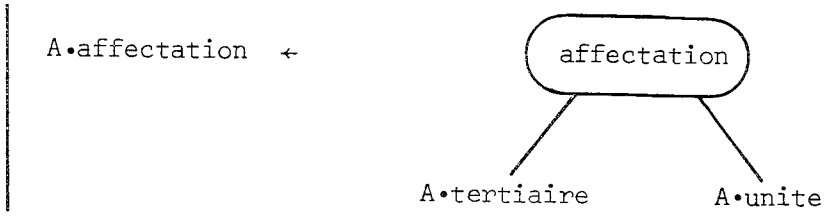


Il reste à construire les accès aux identificateurs, puis les listes de modifications. C'est l'objet des phases suivantes de décoration.

L'arbre abstrait primitif est décrit comme un attribut sur une grammaire hors contexte du langage. Cet attribut, nommé A est synthétisé, ce qui signifie

qu'il est construit à partir des feuilles et que l'arbre complet est attaché à l'axiome de la grammaire. Aux symboles non-terminaux intermédiaires sont attachés des sous-arbres. Ainsi, la règle correspondant à l'affectation s'écrit :

affectation
→ tertiaire, ':=' , unité



L'arbre attaché au non-terminal "affectation" est construit à partir des arbres attachés aux non-terminaux "tertiaire" et "unité" respectivement, grâce à un enracinement par le noeud (affectation) . C'est la signification de la règle d'évaluation associée à la règle syntaxique. Des règles analogues sont associées à "tertiaire" et "unité", et dans le cas de l'exemple considéré les sous-arbres suivants sont synthétisés :

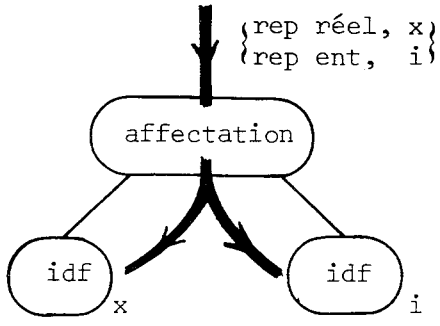
Les informations x et i attachées aux noeuds (idf) représentent en pratique les numéros lexicographiques des identificateurs, ce qui permet lors d'une phase ultérieure de les retrouver dans l'ensemble des déclarations de la *région* et donc de construire la décoration *identificateur* qui représente leur accès. Une *région* en Algol 68 correspond à la notion habituelle de bloc introduite par Algol 60.

Le traitement des déclarations par région et l'identification des identificateurs fait l'objet de la première phase de décoration.

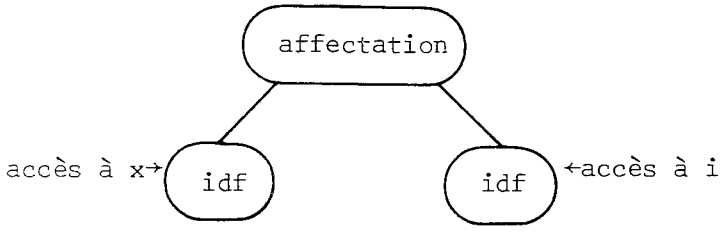
1.2 - Traitement des blocs et identification -

Il s'agit dans cette phase de remplacer les numéros d'identificateur attachés aux noeuds (idf) par la décoration *identificateur* qui représente les caractéristiques de l'identificateur. Or ces caractéristiques doivent être déter-

minées à partir de la déclaration de l'identificateur. Un ensemble d'attributs est utilisé pour décrire la transmission de l'ensemble des déclarations accessibles au niveau de l'utilisation de l'identificateur. Cet ensemble de déclarations est représenté par un attribut hérité. Dans le cas de l'exemple il est constitué des déclarations de la région.



au niveau de chaque noeud (idf) une fonction, *identifier identificateur*, recherche l'identificateur concerné dans l'ensemble de déclarations hérité et remplace le numéro d'identificateur par ses caractéristiques. Après cette phase de décoration, l'arbre obtenu est le suivant :

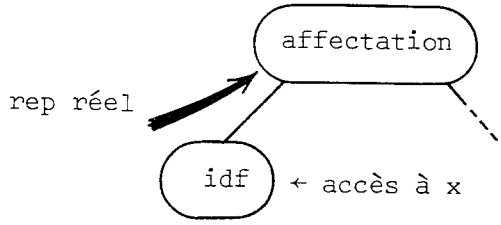


En réalité, l'ensemble des déclarations hérité ne se limite pas aux déclarations de la région mais contient aussi celles des régions englobantes. Ceci est vrai également au niveau du programme principal où sont accessibles non seulement les déclarations du programme mais aussi celles de l'environnement standard qui contient les identificateurs et opérateurs fournis à tous les programmes.

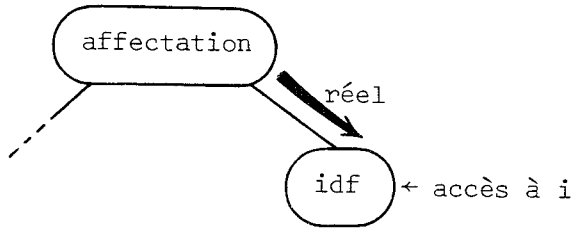
Après cette phase, à chaque feuille de l'arbre est attaché un mode. La deuxième phase de décoration, le traitement des modifications, peut maintenant vérifier que les modes des valeurs utilisées obéissent aux règles du langage,et placer dans l'arbre les modifications nécessaires.

1.3 - Traitement des modifications -

En Algol 68 toute construction a un *mode propre* ou *mode a priori* qui ne dépend que des constituants de la construction. Ainsi, le mode a priori de l'identificateur x est repère de réel, celui de l'identificateur i est repère d'entier et celui de l'affectation est (celui de sa partie gauche) repère de réel. Une fois réalisée l'identification des identificateurs, les modes des feuilles sont connus. Le mode des autres noeuds est déterminé lors du traitement des modifications. Ainsi le mode du noeud (*affectation*) est "synthétisé" depuis son fils gauche , ce qui donne repère de réel

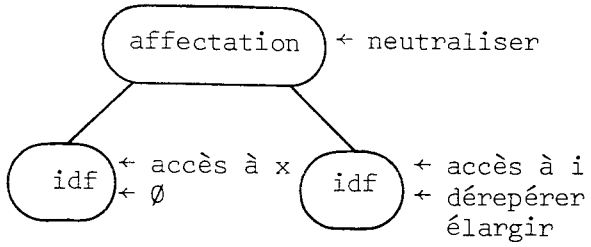


Le mode de la partie droite est obtenu en dérepérant une fois le mode de la partie gauche. C'est donc le mode *réel* qui va être hérité sur le fils droit.



Le mode *réel* est comparé au mode a priori de l'identificateur i : repère d'entier, ce qui donne lieu à la mise en place des modifications *dérepérer* et *élargir*, respectivement.

Le mode a priori de l'affectation, repère de réel, est lui aussi comparé au mode demandé pour cette construction. Dans ce cas, le mode demandé est *neutre*, ce qui signifie que la valeur de l'affectation n'est pas utilisée. Le noeud (affectation) est décoré avec la modification *neutraliser*. On obtient l'arbre abstrait décoré qui va servir à la génération de code.



Remarque : Appliquer à une valeur une modification neutraliser signifie que cette valeur n'est plus utile pour le programme et doit donc être abandonnée. En conséquence cette modification n'entraîne aucune génération de code.

2 - GENERATION DE CODE A PARTIR DE L'ARBRE ABSTRAIT DECORE -

2.1 - Les descripteurs -

Pendant la phase de génération, nous allons décrire chaque valeur qui sera manipulée à l'exécution par un doublet que nous appelons "descripteur".

Chaque descripteur de valeur est formé :

- de la description des caractéristiques de la valeur concernée, c'est-à-dire, son mode et les informations s'y rattachant (exemple : la taille de sa représentation).

- de la description de l'accès à la représentation de la valeur, c'est-à-dire, de l'accès à l'objet interne possédant cette valeur pendant l'exécution. Cet accès est formé de deux informations :

  . le déplacement qu'aura, à l'exécution, la représentation de la valeur, par rapport à une certaine base. Ce déplacement est connu à la compilation.
  . l'adresse de la description de cette base pendant la génération de code.

Une base étant une valeur à part entière, un descripteur de compilation est associé à chacune d'entre elles.

2.2 - Synthèse des descripteurs - Localité de la traduction -

La production effective du code est réalisée au cours d'un parcours de l'arbre abstrait associé à $x := i$.

Le code à générer lors du traitement du noeud (affectation) est fonction des valeurs de la source et de la destination. En conséquence, il est intéressant de disposer, au niveau de ce noeud, des descripteurs de ces deux valeurs. Dans ce but, nous parcourons l'arbre de chacun des fils de ce noeud, avant de produire le code spécifique de l'affectation.

1 - Le parcours du fils gauche (destination) nous conduit à un noeud (identificateur) . A partir de la décoration identificateur nous construisons le descripteur δx associé à x. Ce descripteur peut être représenté ainsi :

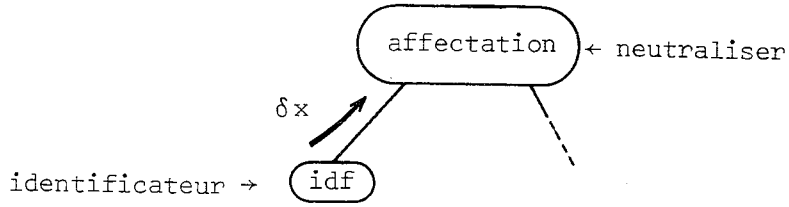
$$\delta x = (C_{repreel}, (d_x, \text{└→}\delta base_x))$$

$C_{repreel}$ représente les caractéristiques d'une valeur de mode *rep réel*.

d_x est le déplacement de l'objet interne correspondant à x, par rapport une base. Cette base, qui peut être commune à beaucoup de valeurs, est décrite par le descripteur $\delta base_x$.

└→ est l'adresse du descripteur $\delta base_x$.

Etant donné qu'il n'y a pas de décoration modification sur le noeud, le descripteur δx décrit la valeur délivrée par la construction associée à ce noeud (x). Il est donc transmis (synthétisé) du noeud (identificateur) (fils) au noeud (affectation) (père).



2 - Le parcours du fils droit (source) aboutit aussi à un noeud (identificateur) . De la même manière, nous construisons le descripteur δi associé à i :

$$\delta i = (C_{repent}, (d_i, \text{└→}\delta base_i))$$

Remarque : Dans le cas présent, $\delta base_x$ et $\delta base_i$ représentent le même descripteur si nous utilisons un système à l'exécution "à la Algol 60".

Des modifications sont à effectuer sur la valeur ainsi décrite.

a) - La modification dérepérer signifie que la valeur à considérer n'est pas la valeur possédée par l'objet interne associé à i, mais celle repérée par cette dernière. En conséquence la valeur décrite par δi va être considérée comme étant une base servant à accéder à la valeur demandée. C'est pourquoi nous construisons un nouveau descripteur δientier utilisant δi comme descripteur de la base. Ce qui donne :

$$\delta\text{ientier} = (C_{ent}, (0, \delta i)) \qquad \delta i = (C_{repent}, (d_i, \delta base_i))$$

C_{ent} représente les caractéristiques d'une valeur entière.

Une modification dérepérer, appliquée à une valeur, ne modifie pas cette valeur, mais permet d'accéder à une autre valeur.

b) - La modification élargir demande, dans cet exemple, de considérer la valeur comme n'étant plus de mode entier mais de mode réel. En théorie la valeur ne change pas et on pourrait penser qu'il suffit de remplacer dans δientier les caractéristiques d'une valeur entière (C_{ent}) par celle d'une valeur réelle ($C_{réel}$). Malheureusement, dans les calculateurs actuels, une valeur est représentée différemment suivant qu'elle est entière ou réelle. A cause de cela, nous produisons du code créant la représentation réelle de la valeur à partir de sa représentation entière.

Pour terminer le traitement il nous reste à construire un descripteur décrivant la valeur réelle ainsi obtenue. En pratique, une valeur créée ainsi, pendant l'exécution, va se retrouver dans une ressource spécialisée (registre, accumulateur, ...) du calculateur. Pour tenir compte de ce fait le descripteur δiréel que nous construisons est d'un type particulier.

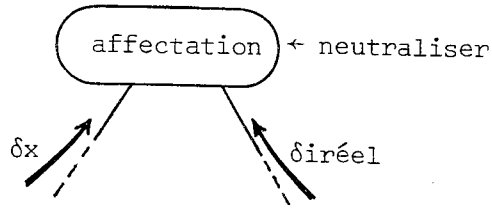
$$\delta\text{iréel} = (C_{réel}, D_{ressα})$$

$D_{ressα}$ est la description de la ressource α contenant la valeur ainsi obtenue.

Le descripteur δientier (la chaîne de descripteurs) décrivant la valeur entière est abandonné puisque du code a été produit pour utiliser cette valeur. De manière générale, l'abandon d'une chaîne de descripteurs entraîne la libération de toutes les ressources spécialisées pouvant y être représentées.

Etant donné que nous avons tenu compte de toutes les décorations modification apparaissant sur le noeud, le descripteur δiréel décrit la valeur délivrée par la construction associée à ce noeud (i). Il est alors synthétisé du noeud (identificateur) (fils) au noeud (affectation) (père).

3 - Les deux fils du noeud (affectation) ayant été traités, les sous-arbres correspondants peuvent être abandonnés. Nous obtenons en définitive le schéma :



L'étape suivante consiste à engendrer la recopie de la représentation de la valeur décrite par δiréel dans l'emplacement dont l'accès est décrit par δx.

Ceci fait, le descripteur δx est considéré comme décrivant le résultat de l'opération d'affectation, puisque la valeur délivrée par une telle opération est celle de la partie gauche (destination). Dans ces conditions, nous abandonnons la chaîne de descripteurs correspondant à la partie droite, ce qui libère la ressource spécialisée contenant la valeur réelle. Puisque la valeur obtenue est à neutraliser - ce qui signifie qu'elle ne sera pas utilisée à l'exécution (il n'y a donc pas de génération de code) - nous abandonnons aussi la chaîne de descripteurs de la partie gauche (x). Le noeud (affectation) est finalement abandonné et aucun descripteur n'est synthétisé vers son noeud père.

2.3 - Le code produit -

Pendant le parcours de l'arbre, le code n'est pas directement produit sous une forme adaptée à un calculateur précis. En pratique, nous avons introduit un niveau intermédiaire que nous décrivons, dans la suite, au moyen de quadruplets [GRIES] indépendants du calculateur. Ce sont ces quadruplets qui sont primitivement produits pendant le parcours de l'arbre.

De manière générale, chaque quadruplet est formé :

- de l'indicateur du traitement à réaliser,
- des opérandes intervenant dans le traitement. Ces opérandes sont soit des descripteurs de valeurs soit des valeurs immédiates.

Pour une opération d'affectation, le quadruplet produit peut se représenter par :

(:=, δdestination, δsource, fant)

où fant indique que l'opérande correspondant est sans signification (fantome)

Pour l'affectation x := i, cela nous donne :

(:=, δx, δiréel,).

Chaque quadruplet ainsi engendré est immédiatement fourni à un traducteur produisant du code chargeable. Ce traducteur a deux tâches :

- produire le code accédant aux représentations des valeurs qui vont être manipulées. Pour cela il utilise les descriptions des fonctions d'accès se trouvant dans les descripteurs opérandes,

- produire le code utilisant ces valeurs en fonction de leurs caractéristiques. Ces informations se trouvent dans les champs C des descripteurs opérandes.

Le code ainsi produit est spécifique du calculateur qui servira à son exécution.

Pour l'affectation $x := i$ à exécuter sur un calculateur IBM 360, supposons que :

- le registre flottant F01 contienne la valeur réelle décrite par δiréel
- le registre général R_b contienne la base décrite par $\delta base_x$.

Dans ces conditions

(:=, δx, δiréel,) est traduit en

STD F01 , $d_x(R_b)$

Par contre, si nous supposons que δiréel décrit une valeur rangée en mémoire au déplacement $d_{iréel}$ par rapport à la base contenue par le registre général R_b, le traducteur réalise les actions suivantes :

- demande d'allocation d'un registre flottant, ce qui met à sa disposition le registre Fn
- production du code

  LD Fn, $d_{iréel}(R_b)$

  STD Fn, $d_x(R_b)$
- libération du registre flottant F_n.

En définitive, dès qu'un quadruplet est produit pendant le parcours de l'arbre, il est transmis au traducteur qui produit le code chargeable correspondant.

Avant de lancer l'exécution d'un programme écrit en Algol 68, il suffit de donner en entrée à un chargeur spécialisé, le code produit par le compilateur.

3 - CONCLUSION -

Sur cet exemple simple nous avons introduit, de manière informelle, les principaux aspects de la méthodologie que nous proposons pour l'écriture d'un compilateur. Remarquons que le code finalement produit est identique à celui obtenu avec des méthodes plus classiques. Une telle méthode de compilation est donc (au moins) aussi bien adaptée que les autres, pour engendrer du code efficace.

II.3 - CONSTRUCTION DE L'ARBRE ABSTRAIT DECORE -

La construction de l'arbre abstrait décoré est décrite en plusieurs étapes : la construction d'un arbre primitif, et des phases de décoration.

La construction de l'arbre abstrait primitif est décrite par un système d'attributs sur une grammaire hors-contexte d'Algol 68. L'emploi de cette grammaire, suppose résolus certains problèmes, ce qui nécessite un travail préalable sur le programme source.

II.3.1 - METHODE -

1 - LE POINT DE DEPART DE LA DESCRIPTION -

La grammaire hors-contexte qui sert de support à la description de l'arbre primitif n'est pas à proprement parler une grammaire d'Algol 68. En effet, nous y trouvons des symboles terminaux comme indicateur de mode ou opérateur prio p qui ne peuvent pas être reconnus par un processus purement lexicographique. En Algol 68 un indicateur est représenté de la même manière que les mots-clé (*début*), les modes standard (*ent*) ou les opérateurs standard (*abs*). Or un indicateur peut servir à spécifier un nouveau mode ou un nouvel opérateur, selon la déclaration qui lui est associée dans une région donnée. Le problème vient du fait qu'une déclaration de variable utilisant un indicateur, et une formule unaire utilisant un indicateur peuvent s'écrire de la même manière.

Considérons le programme Algol 68 suivant :

```
    début
        mode m = ent ;
(1)     m x ;
        ...
        début
            op m = (ent i) ent : i + 1 ;
(2)         m x
        fin
    fin
```

m x représente à la ligne (1) une déclaration de la variable *x* utilisant l'indicateur de mode *m*, et à la ligne (2) une formule utilisant l'opérateur unaire *m*. Rien ne permet de distinguer ces deux constructions si l'indicateur *m* n'a pas été identifié. Or les regrouper dans une même règle afin d'éviter une ambiguïté dans la grammaire diminue considérablement le pouvoir descriptif de cette grammaire. En effet, il s'agit de deux constructions dont les significations sont profondément différentes : une déclaration et une formule.

En écrivant la grammaire de cette manière, nous avons supposé que les deux types d'indicateurs sont différenciés, ce qui nécessite la mise en oeuvre d'un mécanisme d'identification, donc la reconnaissance de la structure de région (structure de bloc). Dans notre implantation cette reconnaissance est réalisée par un passage préalable sur le texte source.

Nous avons également supposé qu'à chaque opérateur est attachée sa priorité, afin de pouvoir exprimer la priorité des formules dans la grammaire. En Algol 68 la priorité des opérateurs peut être changée grâce à une déclaration de priorité, qui a pour portée la région où elle apparaît. Ainsi la formule *a + b * c* a deux significations différentes dans les programmes suivants :

```
début                          début
    a + b * c                      prio + = 7, * = 6 ;
fin                                a + b * c
                                fin
```

programme 1 programme 2

Dans le programme 1 les opérateurs ont leur priorité standard et la formule a son sens habituel : *a + (b * c)*. Dans le programme 2, la priorité des opérateurs + et * a été inversée, ce qui donne à la formule le sens : *(a + b) * c*. Cette possibilité offerte à l'utilisateur trouve sa justification lorsqu'il définit un nouvel opérateur. Il doit également définir sa priorité, qui peut aller de 1 à 9. La priorité des opérateurs unaires est 10 et ne peut être redéfinie. Dans la grammaire que nous proposons, nous considérons pour les opérateurs, 10 symboles terminaux différents : opérateur prio 1, opérateur prio 2, ..., opérateur prio 10. La priorité 10 correspond aux opérateurs unaires. Nous nous plaçons donc dans la situation où les priorités des opérateurs sont reconnues.

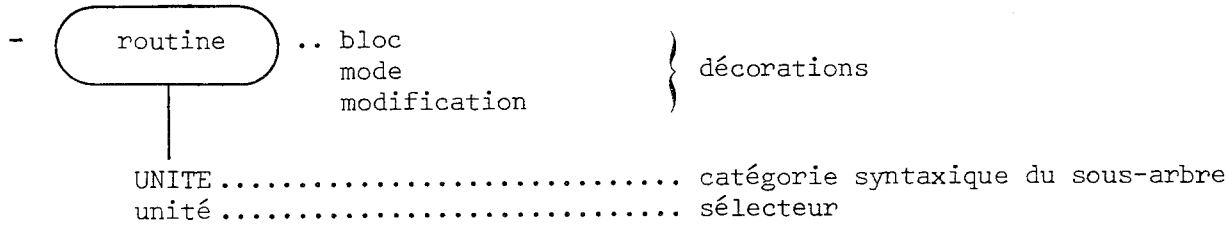
2 - LES ETAPES DE LA DESCRIPTION -

La première étape de la description est la construction de l'arbre abstrait primitif. Cet arbre possède déjà quelques éléments de décoration qui sont utilisés par les phases ultérieures. Ces décorations sont des informations synthétisées dont nous ne décrivons pas la construction. Il aurait été possible de décrire également la construction de ces décorations, mais cela aurait alourdi considérablement la description actuelle. Nous avons préféré insérer un commentaire indiquant leur rôle et la manière dont elles sont construites.

La deuxième étape est la décoration de l'arbre abstrait primitif. Pour des raisons de clarté, cette description est divisée en deux parties : le traitement des blocs et l'identification des identificateurs, puis le traitement des modifications et l'identification des opérateurs. Dans une implantation particulière ces phases peuvent être regroupées ou agencées différemment.

3 - PRESENTATION DES DECORATIONS -

Nous donnons en annexe une grammaire de l'arbre abstrait sur laquelle figurent les décorations qui vont être utilisées lors de la génération de code, ainsi que les sélecteurs associés aux fils de chaque noeud. Nous présentons ici quatre noeuds qui permettent de passer en revue les principales décorations.



exemple de routine : *(ent i, j) bool : i = j*

La décoration mode représente les caractéristiques statiques de la valeur. Elles se déduisent du mode de la valeur, ici *proc (ent, ent) bool* . Il s'agit par exemple de la taille de la partie statique de la représentation.

La décoration bloc indique l'ensemble des caractéristiques des paramètres (mode, niveau statique, déplacement dans la zone des variables locales ...).

Elle correspond à un ensemble de décorations identificateur.

La décoration modification est formée de la liste - éventuellement vide - des modifications à appliquer au noeud.

Sur le noeud (routine) , les décorations mode et bloc représentent des informations qui étaient présentes dans le texte mais qui n'apparaissent plus dans l'arbre.

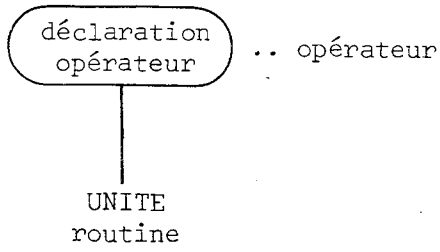
- (définition idf) .. identificateur
  |
  UNITE-option initialisation

Le noeud (définition-idf) apparaît dans les déclarations d'identificateur : déclaration d'identité et déclaration de variable. Ainsi, pour la déclaration *ent i, j* seront créés deux noeuds (définition-idf) .

La décoration identificateur contient le mode de l'identificateur, ainsi que toutes les informations concernant l'accès à la partie statique de la représentation de la valeur possédée par l'identificateur (niveau statique, déplacement dans la zone des variables locales, etc...). On retrouve cette décoration identificateur sur le noeud (idf) après la première phase de décoration de l'arbre abstrait : l'identification des identificateurs.

- (déclaration mode) .. indicateur
  |
  DECLAREUR déclareur

Une déclaration de mode associe un mode spécifié par un déclareur, à un indicateur (*mode ligne* =[*1 : n*]*car*). La décoration indicateur caractérise la routine qui va être créée pour le déclareur (voir le traitement du noeud (déclaration-mode) au chapitre II.4.6. § 4.2).



Une déclaration d'opérateur associe un texte de routine à un opérateur (*op égale = (ent i, j) bool : i = j*). La décoration opérateur est semblable à la décoration identificateur pour une procédure.

Les autres décorations sont introduites au moment de leur utilisation. On en trouve une liste récapitulative en annexe.

II.3.2 - CONSTRUCTION DE L'ARBRE ABSTRAIT DECORE -

1 - PRINCIPE -

Etant donné une grammaire hors-contexte d'Algol 68 et un attribut A attaché à chaque non-terminal de la grammaire, l'arbre abstrait du programme est la représentation de l'attribut A attaché à la racine. Il est synthétisé depuis les feuilles vers la racine. A chaque membre de partie droite d'une règle de la grammaire est associée une règle de synthèse de l'arbre abstrait attaché au non-terminal de gauche. Il est fonction des arbres attachés aux non-terminaux de la partie droite de la règle. Ce processus ne met pas en oeuvre d'attributs hérités.

2 - ECRITURE DE LA GRAMMAIRE -

1 - La forme d'une règle de la grammaire est la suivante :

non-terminal
→ alternance
| règles d'évaluation des attributs pour cette alternance
→ alternance
| ...
...

Une alternance est une séquence de symboles terminaux et non-terminaux séparés par des virgules. Les symboles terminaux sont soulignés ou entre apostrophes. Les symboles non-terminaux peuvent être indicés s'il faut les différencier pour l'évaluation des attributs.

Lorsque l'on a le choix entre plusieurs symboles terminaux ils sont indiqués entre accolades.

Exemple :

$$x \rightarrow \left\{\begin{matrix} \textit{début} \\ \text{'('} \end{matrix}\right\}, y, \left\{\begin{matrix} \textit{fin} \\ \text{')'} \end{matrix}\right\}$$

est équivalent aux règles :

$x \rightarrow$ *début* , y, *fin*

$x \rightarrow$ '(' , y, ')'

2 - L'attribut x attaché au non-terminal n est noté **x.n**
La règle d'évaluation de l'attribut **x.n** s'écrit :
x.n ← une expression décrivant le calcul de l'attribut

3 - Transmission par identité .
Lorsqu'une règle a un seul non-terminal en partie droite et que l'attribut A est transmis sans modification on note cette règle :

partie gauche ⊸ **partie droite**

Ainsi la règle

$$y \rightarrow \underline{\alpha}, z \mid A.y \leftarrow A.z$$

où $\underline{\alpha}$ est un symbole terminal, est notée :

$$y \multimap \underline{\alpha}, z$$

4 - Conventions particulières
⊕ représente la concaténation d'arborescences [PAIR]
Λ représente l'élément neutre de la concaténation d'arborescences.
On utilise les abréviations x-liste et x-option, où x est un non-terminal quelconque. Les règles correspondantes sont les suivantes :

x-option → ∅
| A.x-option ← Λ

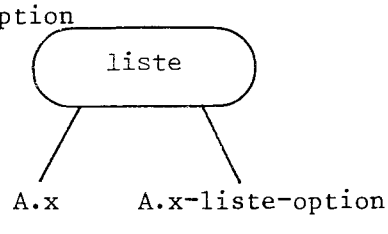
→ x
| A.x-option ← A.x

x-liste → x, x-liste-option
| A.x ← A.x ⊕ A.x-liste-option

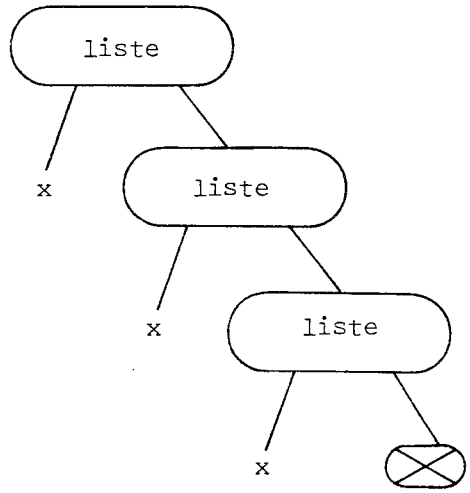
ce qui crée des arborescences du type x ⊕ x ⊕ x ⊕ ....

Une représentation alternative est fournie par :

x-liste → x, x-liste-option
| A.x-liste ←



ce qui crée des arborescences du type



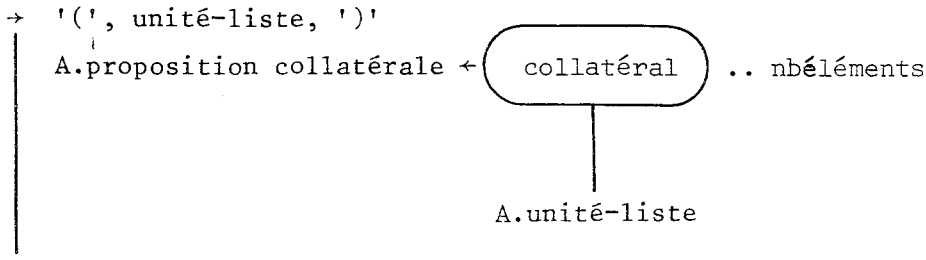
 représente le pointeur de fin de liste.

5 - Mise en place de décorations.

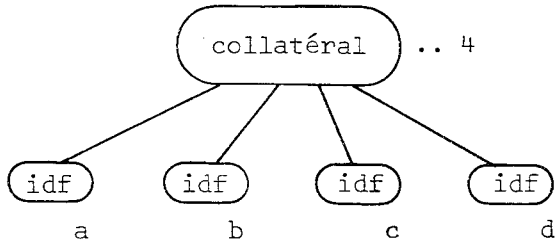
Certains éléments de décoration sont mis en place lors de la construction de l'arbre abstrait. Ces décorations sont notées à côté du noeud, avec le symbole .. .

Une proposition collatérale est formée d'une liste d'unités. La règle correspondante s'écrit :

proposition collatérale



La décoration nbéléments représente le nombre d'unités de la liste. Ainsi, dans le cas de la proposition collatérale *(a, b, c, d)* l'arbre créé est :

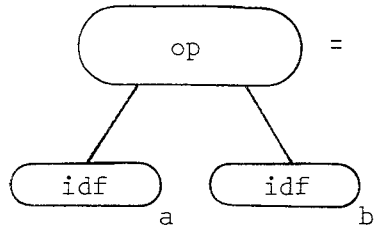


3 - EXEMPLE -

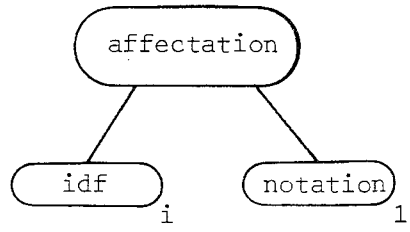
Considérons la proposition conditionnelle :

si a = b alors i := 1 sinon p(i) fsi

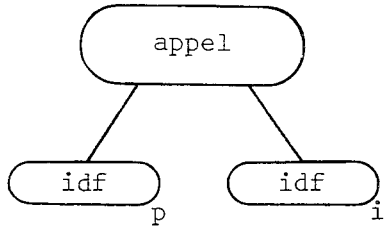
L'arbre associé à la condition *a = b* est :



l'arbre associé à la partie alors *i := 1* est :



l'arbre associé à la partie sinon *p(i)* est :

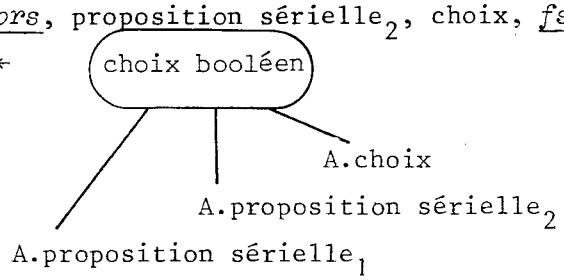


l'arbre associé à la proposition conditionnelle est construit en fonction de ces sous-arbres. Sa construction est décrite par la règle :

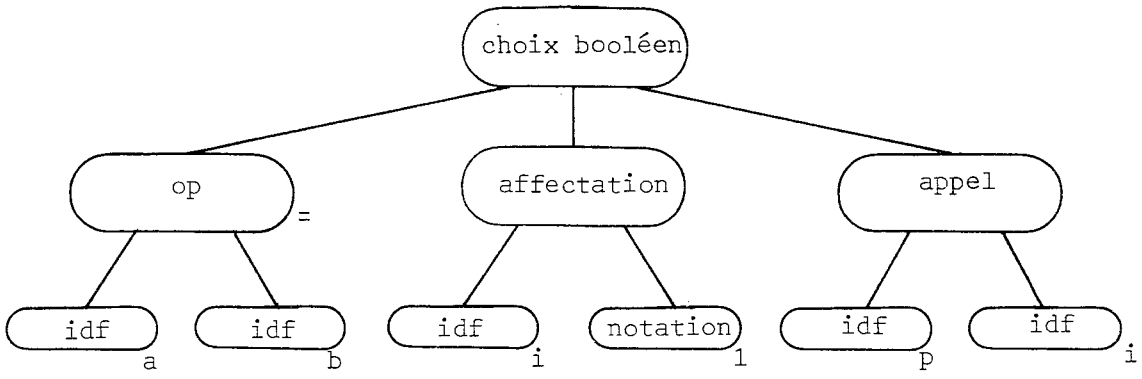
proposition conditionnelle

→ *si*, proposition sérielle$_1$, *alors*, proposition sérielle$_2$, choix, *fsi*

A.proposition conditionnelle ←



Dans le cas de l'exemple, "proposition sérielle$_1$" est réduit à l'expression *a = b*, "proposition sérielle$_2$" à l'affectation *i := 1* et "choix" à l'appel *p(i)*. L'arbre construit est donc

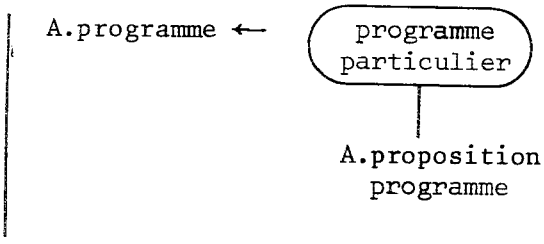


Nous donnons maintenant la grammaire et les règles de construction de l'arbre abstrait primitif.

4 - LA GRAMMAIRE ET LES REGLES DE CONSTRUCTION DE L'ARBRE PRIMITIF -

programme

⟶ proposition



proposition

⟶0⟶ proposition fermée

⟶0⟶ proposition collatérale

⟶0⟶ proposition conditionnelle

⟶0⟶ proposition cas entier

⟶0⟶ proposition cas union

⟶0⟶ proposition itérative

proposition fermée

⟶ { *début* / '(' } , proposition sérielle, { *fin* / ')' }

si contient des déclarations (A.proposition sérielle)

alors A.proposition fermée ← début

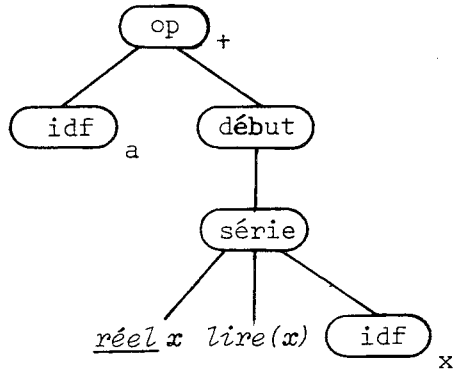
A.proposition sérielle
série

sinon A.proposition fermée ← A.proposition sérielle

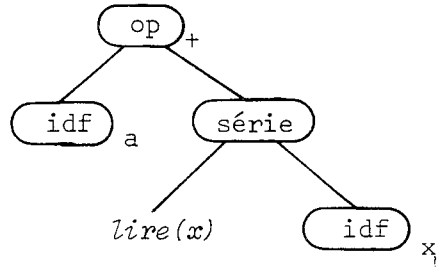
finsi

co Le noeud (début) n'est créé que lorsqu'il y a des déclarations dans le bloc. Sinon, il s'agit d'un simple parenthésage, analogue à celui des expressions arithmétiques dans d'autres langages.
L'arbre créé pour *a +(réel x ; lire(x) ; x)*

contient un noeud (début)



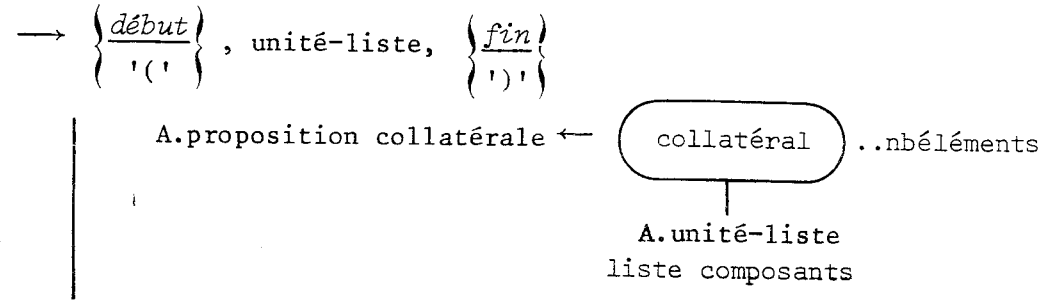
l'arbre créé pour *a +(lire(x) ; x)*
n'en contient pas



La fonction "contient des déclarations" qui s'applique à l'arbre synthétisé pourrait être remplacée par la synthèse d'un attribut booléen supplémentaire qui indique la présence de déclarations.

co

proposition collatérale

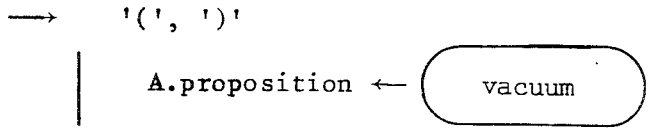


co exemples de proposition collatérale :

début lire(i), écrire (j) fin
(a, b, c, d)

La décoration nbéléments indique le nombre d'unités constituant la proposition collatérale.

co



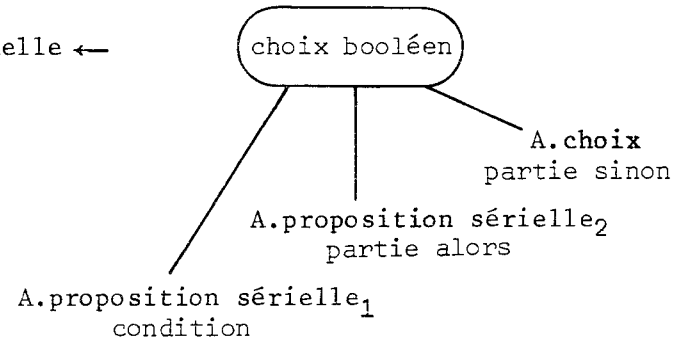
co tableau vide co

proposition conditionnelle

$\longrightarrow$ { *si* / '(' } , proposition sérielle$_1$, { *alors* / '|' } , proposition sérielle$_2$,

choix, { fsi / '(' }

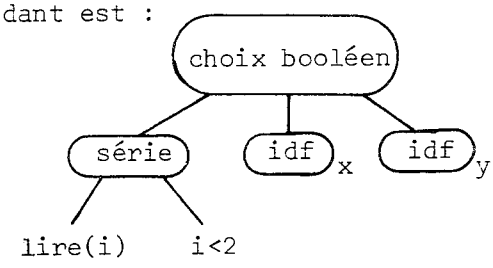
A.proposition conditionnelle ←



co exemple de proposition conditionnelle

si lire(i) ; i < 2 alors x sinon y fsi

L'arbre correspondant est :



co

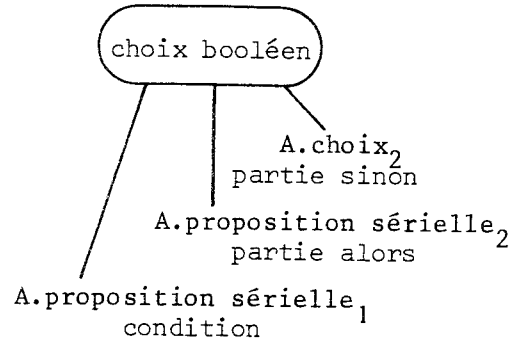
choix$_1$

⟶ { *sinon* / '|' } , proposition sérielle

| A.choix ← A.proposition sérielle

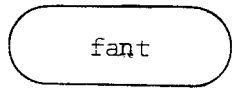
⟶ { *sinsi* / '|' } , proposition sérielle$_1$, { *alors* / '|' } , proposition sérielle$_2$,
choix$_2$

| A.choix$_1$ ←

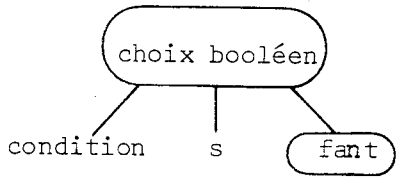


⟶ ∅

| A.choix ←



co L'arbre créé pour *si condition alors s fsi* est



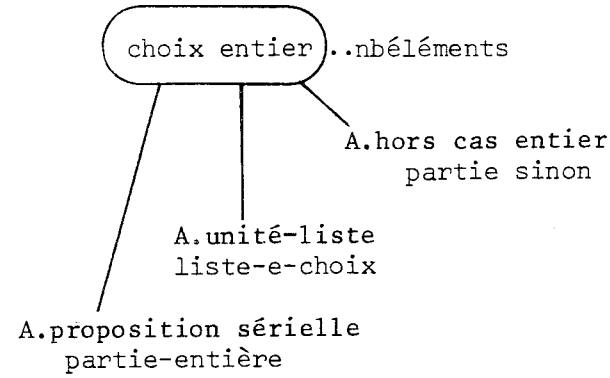
co

proposition cas entier

⟶ {*cas* / '('} , proposition sérielle, {*dans* / '|'} , unité-liste,

hors cas entier, {*fcas* / ')'}

A.proposition cas entier ←



hors cas entier$_1$

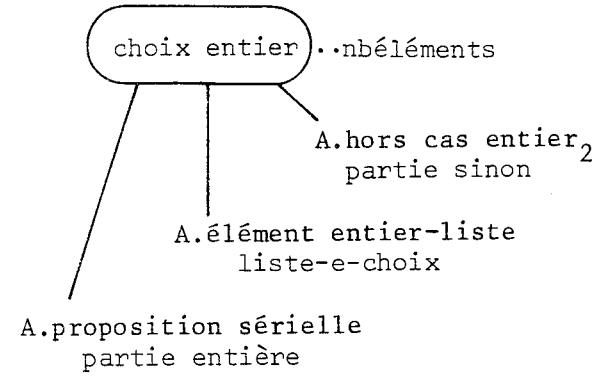
⟶ {*sinon* / '|'} , proposition sérielle

A.hors cas entier ← A.proposition sérielle

⟶ {*sinsi* / '|:'} , proposition sérielle, {*dans* / '|'} , élément entier-liste,

hors cas entier$_2$

A.hors cas entier$_1$ ←



⟶ ∅

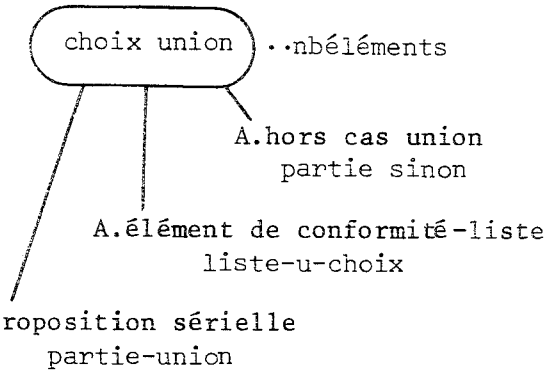
A.hors cas entier ←

proposition cas union

→ {*cas* / '('} , proposition sérielle, {*dans* / '|'} , élément conformité-liste,

hors cas union, {*fcas* / ')'}

A.proposition cas union ←



hors cas union$_1$

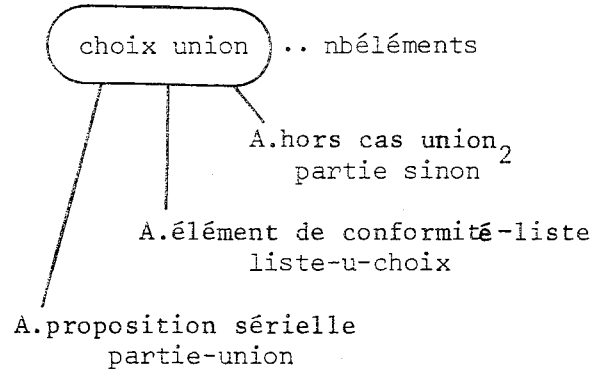
→ {*hors* / '|'} , proposition sérielle

A.hors cas union ← A.proposition sérielle

→ {*sinsi* / '|:'} , proposition sérielle, {*dans* / '|'} , élément de conformité-liste,

hors cas union$_2$

A.hors cas union$_1$ ←



→ ∅

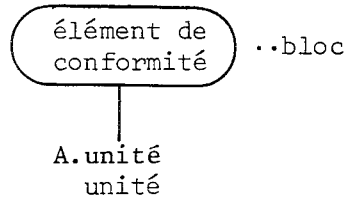
A.hors cas union ← fant

élément de conformité

⟶ '(', déclareur formel, identificateur-option, ')',

':', unité

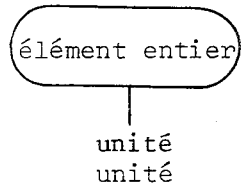
A.élément de conformité ←



co La décoration bloc correspond à l'identificateur optionnel qui suit le déclareur formel. co

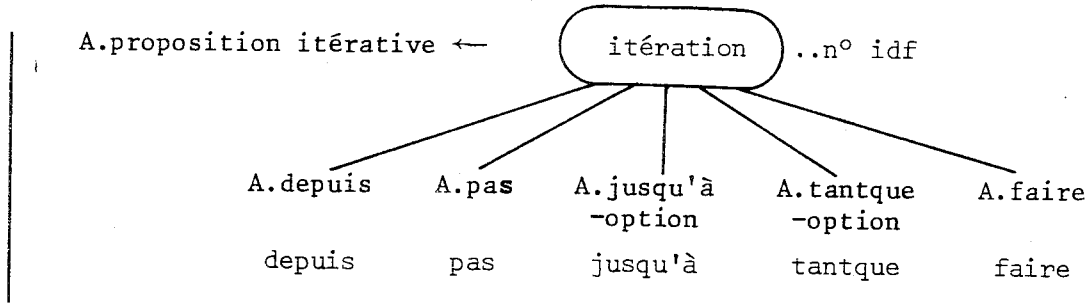
élément entier ⟶ unité

A.élément entier ←



proposition itérative

⟶ pour-option, depuis-option, pas-option, jusqu'à-option, tantque-option, faire



pour

⟶ *pour*, identificateur

⟶ ∅

co La décoration n° idf correspond à l'identificateur de la "partie pour", pour lequel il n'est pas créé d'arbre. co

depuis

⟶ *depuis*, unité

⟶ ∅

co 1 est la valeur par défaut lorsque la "partie depuis" est absente. co

pas

⟶ *pas*, unité

⟶ ∅

co 1 est la valeur par défaut lorsque la "partie pas" est absente

pour i jusqu'à 100 faire p fait

a la même signification que

pour i depuis 1 pas 1 jusqu'à 100 faire p fait

co

jusqu'à

—0→ *jusqu'à*, unité

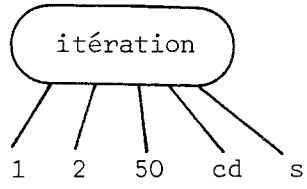
tantque

—0→ *tantque*, proposition sérielle

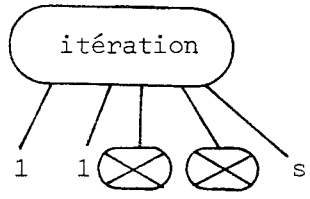
faire

—0→ *faire*, proposition sérielle, *fait*

co L'arbre créé pour l'instruction
pour i depuis 1 pas 2 jqa 50 tantque cd faire s fait
est :



L'arbre créé pour *faire s fait* est :

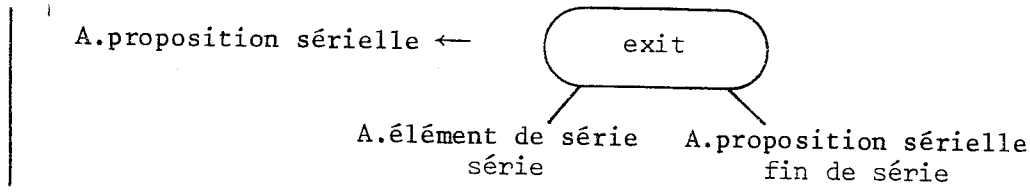


co

proposition sérielle

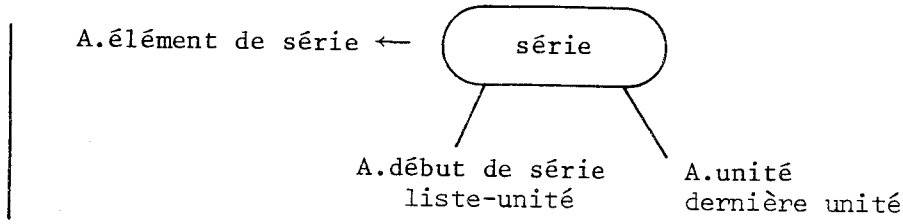
⊸ élément de série

⟶ élément de série, *exit*, proposition sérielle



élément de série

⟶ début de série, ';', unité



⊸ unité

co Il n'est créé de noeud (série) que lorsqu'il y a plus d'une unité. La dernière unité est alors différenciée de celles qui précèdent, car c'est elle qui détermine le mode de la proposition sérielle. Dans l'affectation *x := (lire(y) ; y)* c'est la valeur de la dernière unité, *y* qui est affectée à x. co

début de série$_1$

⟶ déclaration, ';', début de série$_2$

| A.début de série$_1$ ← A.déclaration ⊕ A.début de série$_2$

⟶ unité étiquetable, ';', début de série$_2$

| A.début de série ← A.unité étiquetable ⊕ A.début de série$_2$

⟼ déclaration

⟼ unité étiquetable

co L'arbre créé pour début de série a la forme d'une liste (concaténation de sous arbres). co

unité étiquetable$_1$

⟼ unité

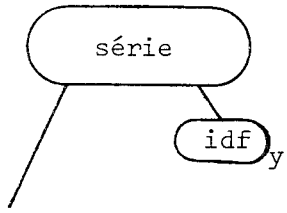
⟶ étiquette, unité étiquetable$_2$

| A.unité étiquetable$_1$ ← étiquette

A.unité étiquetable$_2$
unité

co exemple d'arbre créé pour une proposition sérielle

reel x,y ; lire((x,y)) ; x := x+y ; y



reel x,y ; lire((x,y)) ; x := x+y

La liste d'unités et de déclarations qui constitue le fils gauche du noeud (série) est représentée ici sous la forme d'une liste binaire. Dans notre implantation nous avons utilisé un procédé de segmentation de l'arbre qui permet de les traiter au même niveau. Cet aspect est traité au chapitre III.3.

co

déclaration

⊸ déclaration simple

⊸ déclaration collatérale

déclaration collatérale$_1$

⟶ déclaration simple$_1$, ',', déclaration simple$_2$

A.déclaration collatérale ← A.déclaration simple$_1$ ⊕
A.déclaration simple$_2$

⟶ déclaration simple, ',', déclaration collatérale$_2$

A.déclaration collatérale ← A.déclaration simple ⊕
A.déclaration collatérale

déclaration simple

⊸ déclaration de mode
⊸ déclaration d'opérateur
⊸ déclaration de priorité
⊸ déclaration d'identité
⊸ déclaration de variable

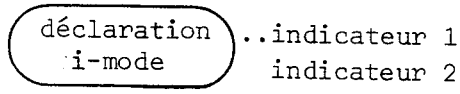
déclaration de mode

⟶ *mode*, *indicateur de mode*, '=', déclareur actuel

A.déclaration de mode ←

si le déclareur est un indicateur contenant des parties dynamiques

alors



co Exemple de déclaration :

mode indicateur 1 = *indicateur 2* ;

où

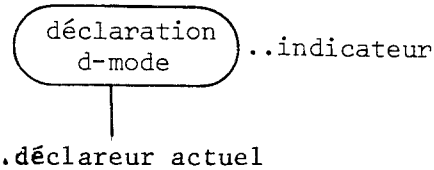
mode indicateur 2 = [*a*:*b*] *ent*

Si *indicateur 2* est un indicateur représentant un mode sans parties dynamiques (*mode indicateur 2* = *rep ent*), la déclaration du mode *indicateur 1* n'apparaît pas dans l'arbre car il n'y a aucun travail de génération de code à faire.

co

sinon si le déclareur contient des parties dynamiques

alors

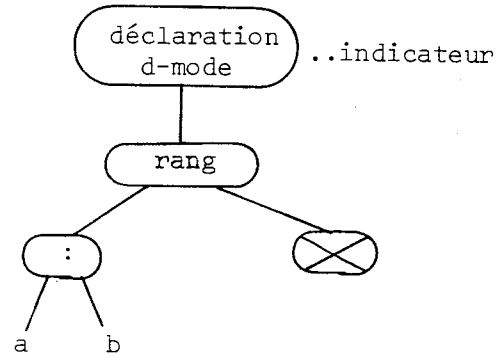


A.déclareur actuel
déclareur

co Exemple de déclaration de mode faisant intervenir des parties dynamiques, c'est-à-dire un tableau :

mode indicateur = [*a*:*b*] *ent*

L'arbre correspondant est

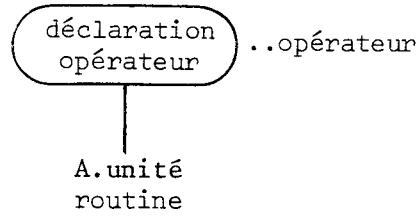


```
        Le mode des éléments du tableau, ent, n'apparaît pas dans
        l'arbre. Il est contenu dans la décoration indicateur.
    co
sinon Λ
    co Lorsque le déclareur ne fait pas intervenir de tableau en
        position actuelle, c'est-à-dire avec des bornes, il n'y a pas
        d'arbre correspondant.
    co
finsi finsi
```

déclaration d'opérateur

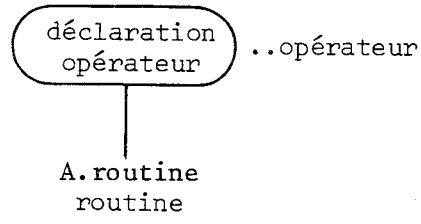
⟶ *op*, plan virtuel, opérateur, '=', unité

A.déclaration d'opérateur ⟵



⟶ *op*, opérateur, '=', texte de routine

A.déclaration d'opérateur ⟵



co La décoration opérateur contient le mode de l'opérateur déclaré. Celui-ci est donné par le "plan virtuel" quand la déclaration est du type : *op (ent, ent) bool p = ...* .
S'il s'agit d'une déclaration sans "plan virtuel" le mode de l'opérateur est déduit de la routine :

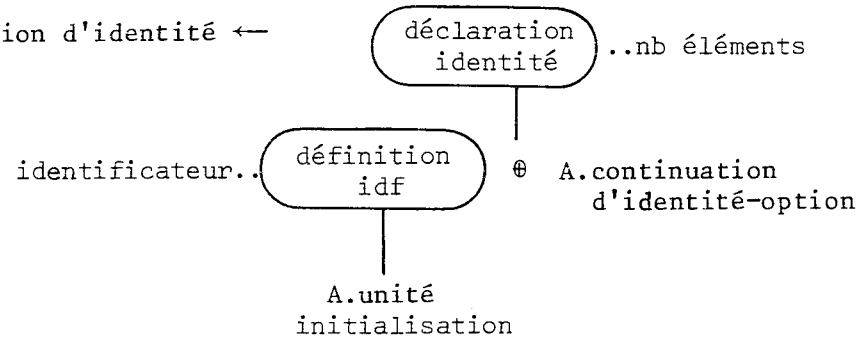
op p = (ent i,j) bool : ...

co

déclaration d'identité

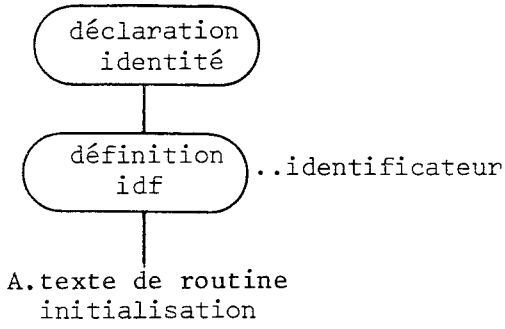
⟶ déclareur formel, identificateur, '=', unité,
continuation d'identité-option

A.déclaration d'identité ←



⟶ *proc*, identificateur, '=', texte de routine,
identité proc cont-option

A.déclaration d'identité ←

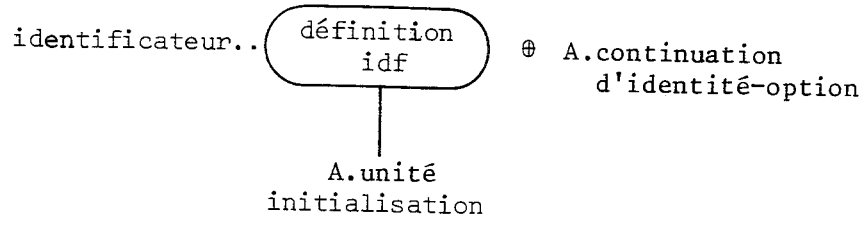


co La déclaration d'identité établit un lien direct entre l'identificateur déclaré et l'objet interne associé.
Exemple : *reel pi = 3.14*
La valeur de *pi* ne peut plus être changée dans la portée de la déclaration.
co

continuation d'identité

⟶ ',', identificateur, '=', unité, continuation d'identité-option

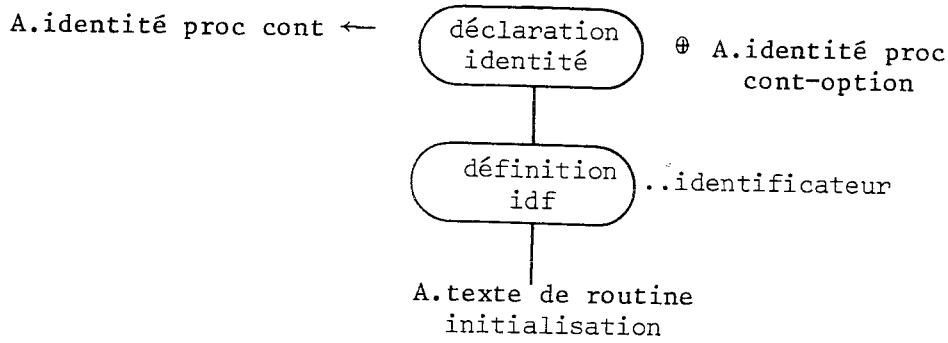
A.continuation d'identité ←



identité proc cont

⟶ ',', identificateur, '=', texte de routine,

identité proc cont-option



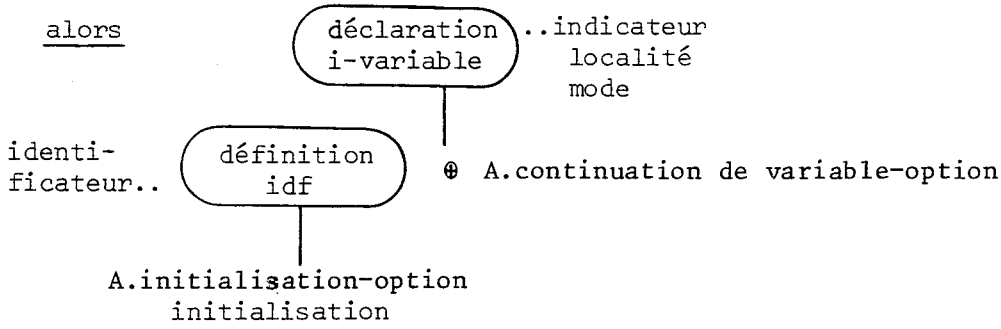
déclaration de variable

⟶ {*loc* | *tas*} -option, déclareur actuel, identificateur, initialisation-option, continuation de variable-option

A.déclaration de variable

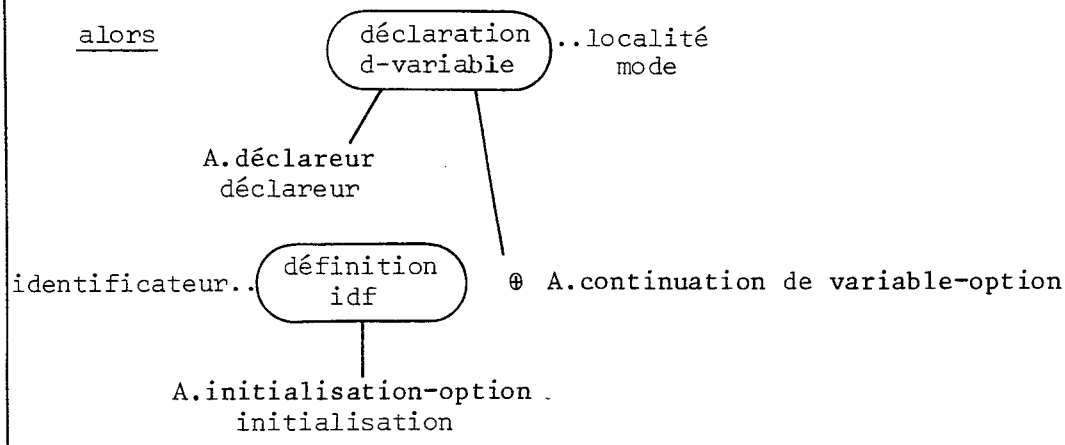
si le déclareur est un indicateur contenant des parties dynamiques

alors

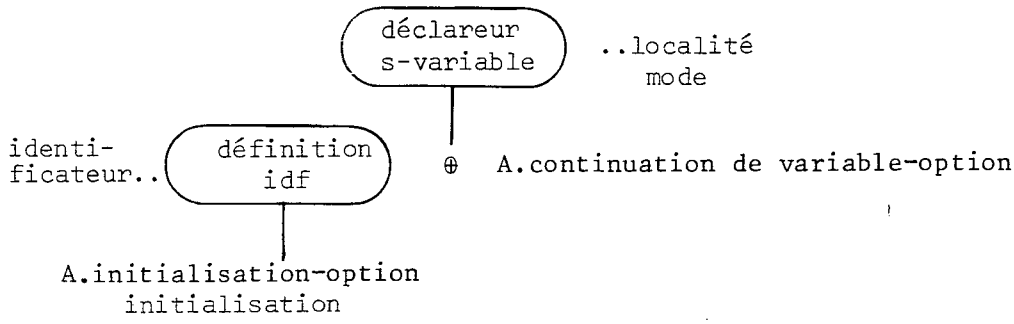


sinon si le déclareur contient des parties dynamiques

alors



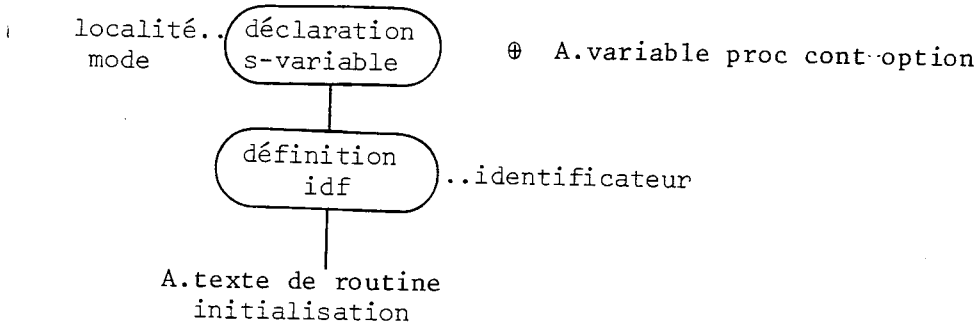
sinon co déclareur simple co



finsi finsi

⟶ {*loc* | *tas*} -option, *proc*, identificateur, ':=',
texte de routine, variable proc cont-option

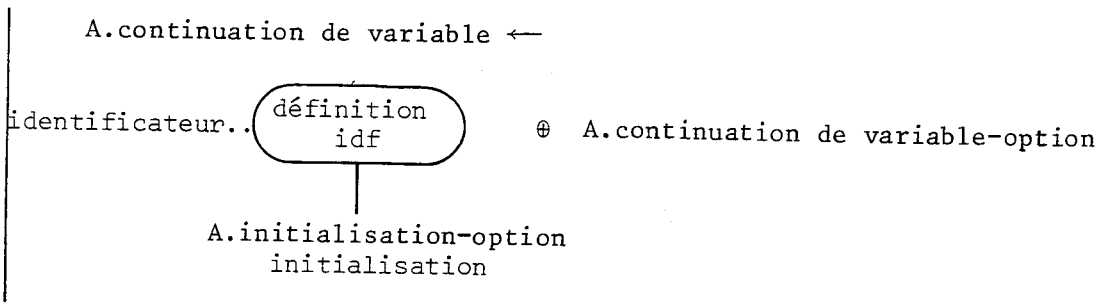
A.déclaration de variable ←



continuation de variable

⟶ ',', identificateur, initialisation-option,
continuation de variable-option

A.continuation de variable ←



initialisation

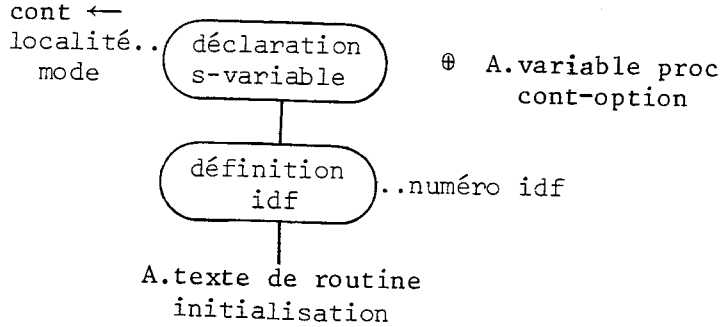
⟶ ':=', unité

A.initialisation ← A.unité

variable proc cont

⟶ ',', identificateur, ':=', texte de routine,
variable proc cont-option

A.variable proc cont ←



unité

—0→ affectation

—0→ relation d'identité

—0→ texte de routine

—0→ saut

—0→ fantôme

—0→ tertiaire

tertiaire

—0→ formule

—0→ nihil

—0→ secondaire

secondaire

—0→ générateur

—0→ sélection

—0→ primaire

primaire

—0→ tranche

—0→ appel

—0→ forceur

—→ notation

| A.primaire ← (notation) ..type

—→ identificateur

| A.primaire ← (idf) ..n° idf

—0→ texte de format

—0→ proposition

affectation

⟶ tertiaire, ':=', unité

A.affectation ← affectation

affectation
- A.tertiaire destination
- A.unité source

relation d'identité

⟶ tertiaire$_1$, { *est* / *n'est pas* }, tertiaire$_2$

A.relation d'identité ← relation identité

relation identité
- A.tertiaire$_1$ opérande gauche
- A.tertiaire$_2$ opérande droit

texte de routine

⟶ plan formel, ':', unité

A.texte de routine ← routine ..bloc

routine
- A.unité unité

co Le plan formel d'une routine contient la spécification des modes des paramètres. Cette information se retrouve dans la décoration bloc du noeud (routine)

Exemple : *(ent i,j) bool : i=j*

co

plan formel

⟶ déclareur formel

⟶ '(', champ formel-liste, ')', déclareur formel

champ formel

⟶ déclareur formel, identificateur

co Il n'y a pas d'arbre correspondant à un déclareur formel car il ne contient pas de bornes. co

saut

⟶ *aller à* -option, identificateur

A.saut ← (saut) ..numéro idf

fantôme

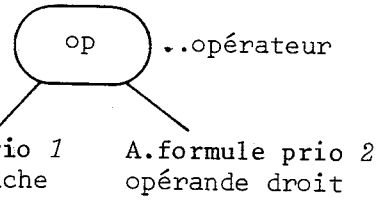
⟶ *fant*

A.fantôme ← (fant)

formule

⟶ formule prio *1*, opérateur prio *1*, formule prio *2*

A.formule ⟵



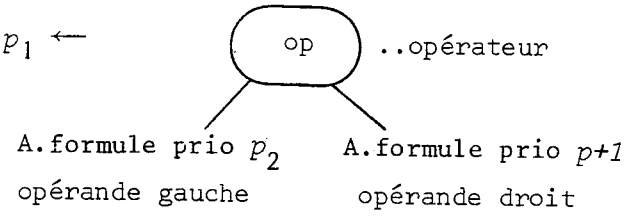
⟶ formule prio *2*

p de 1 à 9 :

formule prio p_1

⟶ formule prio p_2, opérateur prio p, formule prio $p+1$

A.formule prio p_1 ⟵

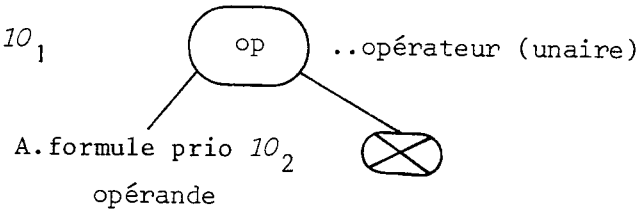


⟶ formule prio $p+1$

formule prio 10_1

⟶ opérateur prio *10*, formule prio 10_2

A.formule prio 10_1



⟶ secondaire

nihil

⟶ nil

A.nihil ⟵

générateur

⟶ { *loc* / *tas* } , déclareur actuel

A.générateur ←

si le déclareur est un indicateur contenant des parties dynamiques

alors (i-générateur) ..indicateur
localité
mode

sinon si le déclareur contient des parties dynamiques

alors (d-générateur) ..localité
mode

A.déclareur
déclareur

sinon co déclareur simple co

(s-générateur) ..localité
type

finsi finsi

sélection

⟶ identificateur, *de*, secondaire

A.sélection ← (sélection) ..champ

A.secondaire
secondaire

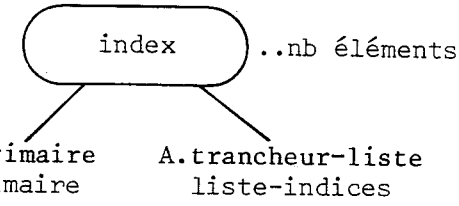
tranche

⟶ primaire, {'('/'['} , trancheur-liste, {')'/']'}

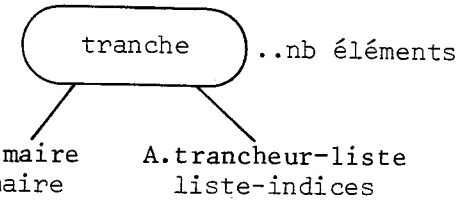
A.tranche ←

si tous les trancheurs sont des unités

alors



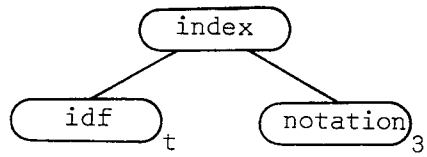
sinon



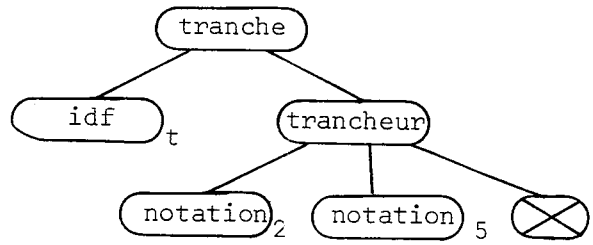
finsi

co Exemples de tranche et arbres correspondants

t [3]



t [2:5]

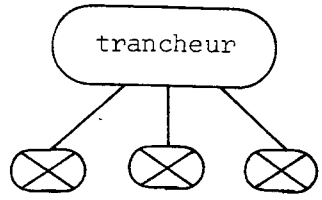


co

trancheur

⟶ ∅

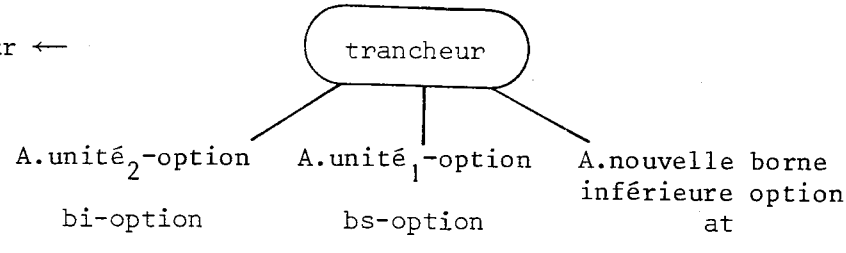
A.trancheur ⟵



⟶0⟶ unité

⟶ unité$_1$-option, ':', unité$_2$-option,
nouvelle borne inférieure-option

A.trancheur ⟵



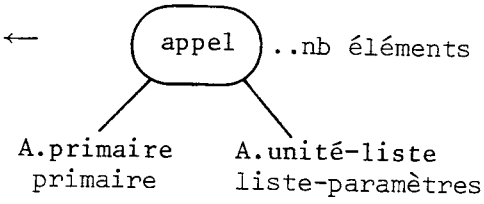
nouvelle borne inférieure

⟶0⟶ *apd*, unité

appel

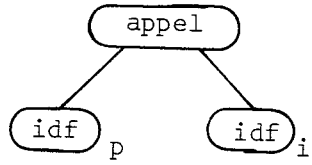
⟶ primaire, '(', unité-liste, ')'

A.appel ←

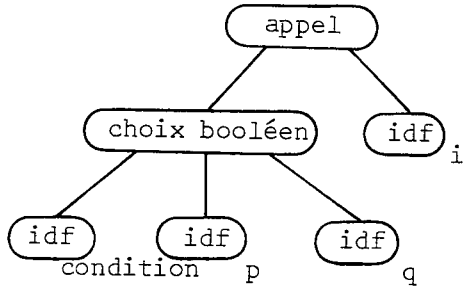


co Exemples d'appels et arbres correspondants :

p(i)



si condition alors p sinon q fsi (i)

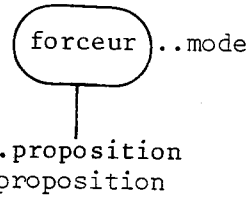


co

forceur

⟶ déclareur formel, proposition

A.forceur ←



déclareur$_1$

⟶ indicateur de mode

A.déclareur ← (indicateur) ..duplication indicateur

⟶ {'('; '['} , bornes-liste, {')'; ']'} , déclareur$_2$

A.déclareur$_1$ ← (rang) ..dimension duplication

A.bornes-liste liste-bornes

A.déclareur$_2$ déclareur

⟶ *struct*, '(', champ-liste, ')'

A.déclareur ←

si A.champ-liste ≠ Λ

alors (struct) ..champs duplication

A.champ-liste liste-champs

sinon Λ

finsi

⟶ *rep*, déclareur

A.déclareur ← Λ

⟶ *proc*, déclareur

A.déclareur ← Λ

⟶ *proc*, '(', déclareur-liste, ')', déclareur

A.déclareur ← Λ

⟶ *union*, '(', déclareur-liste, ')'

A.déclareur ← Λ

⟶ mode primitif

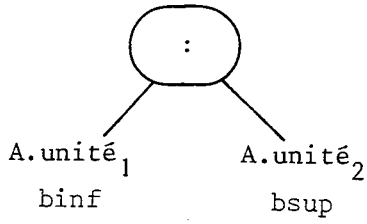
A.déclareur ← Λ

bornes

—0→ unité

—0→ unité$_1$, ':', unité$_2$

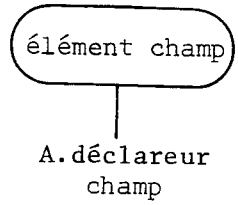
A.bornes ←



champ

⟶ déclareur, identificateur-liste

A.champ ←



co L'arbre d'un déclareur permet de retrouver ses bornes et de construire la routine associée en explorant parallèlement le mode du déclareur.

A un sous déclareur qui ne contient pas de bornes correspond un arbre vide.

co

5 - REMARQUES -

L'identification des identificateurs, le traitement des modes et l'identification des opérateurs ne peuvent être exprimés par une grammaire hors-contexte. Ils sont décrits dans le rapport de définition du langage par une grammaire à deux niveaux [VANWIJ-3]. Un tel mécanisme n'est pas directement implantable, [CHAS-COL] mais nous nous en sommes inspirés pour la description des phases de décoration de l'arbre, qui sont exprimées par un système d'attributs sur l'arbre abstrait.

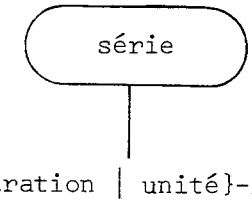
II.3.3 - PREMIERE PHASE DE DECORATION : TRAITEMENT DES BLOCS ET IDENTIFICATION DES IDENTIFICATEURS -

1 - PRINCIPE -

Les règles concernant la portée des identificateurs dans un langage à structure de bloc peuvent s'exprimer à l'aide d'un mécanisme d'attributs. Considérons l'arbre abstrait associé à une construction définissant une région (un bloc). Un tel sous arbre a un noeud père ((début) , (choix-booléen) , ...), et les déclarations de la région font partie de ses opérandes. A toute utilisation d'un objet en dehors de sa déclaration, il faut associer ses caractéristiques, qui sont exprimées précisément dans sa déclaration. Ce processus de rattachement d'un objet à sa déclaration s'appelle identification. Il nécessite la connaissance des ensembles de déclarations de la région et des régions englobantes.

Pour chaque région, l'ensemble des déclarations est synthétisé jusqu'au noeud père de la région. Cet ensemble est représenté par l'attribut SBLOC. Sa construction est décrite au niveau du noeud (série) qui correspond à toute proposition sérielle de plus d'un composant, ce qui est le cas lorsqu'il y a des déclarations.

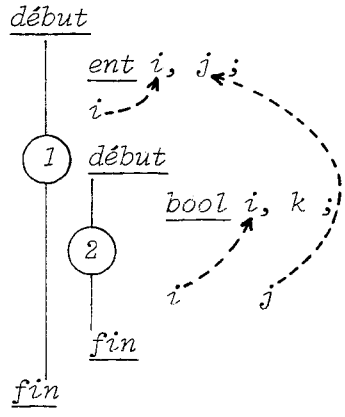
$$SBLOC.série \leftarrow \bigcup_i (SBLOC.déclaration_i)$$

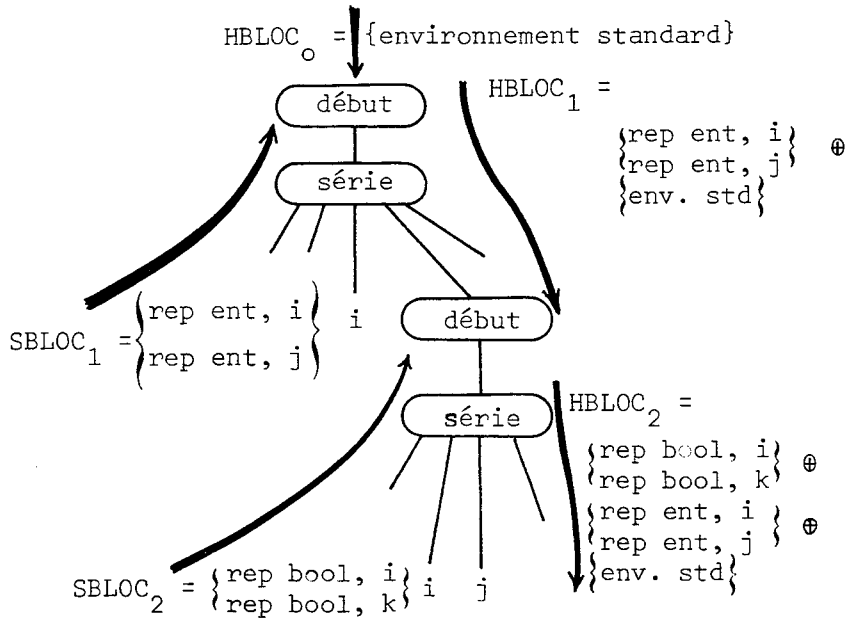


l'attribut SBLOC étant construit au niveau de chaque déclaration et contenant par exemple pour un identificateur son mode et son nom.

Une fois déterminé cet ensemble de déclarations pour une région, il faut les rendre accessibles jusqu'au niveau des occurrences d'utilisation des objets, c'est-à-dire jusqu'aux noeuds du sous-arbre de la région. C'est le rôle de l'attribut hérité HBLOC. Au niveau de chaque noeud délimitant une région, est constitué un nouvel ensemble HBLOC. Il est formé de l'ensemble ordonné des déclarations valides dans la région où il va descendre. Cet ensemble n'est pas seulement constitué des déclarations propres à la région, mais aussi des déclarations des régions englobantes, qui ont été descendues jusqu'à ce noeud par le mécanisme d'héritage des attributs.

Exemples :

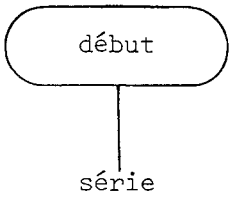




L'identification d'un identificateur consiste alors à le rechercher dans l'attribut HBLOC qu'il reçoit et à le décorer en conséquence. C'est le rôle de la fonction *identifier* :

fonction *identifier* = (no idf, HBLOC) identificateur :
co retourne les caractéristiques de l'identificateur :
décoration identificateur co
finfonction

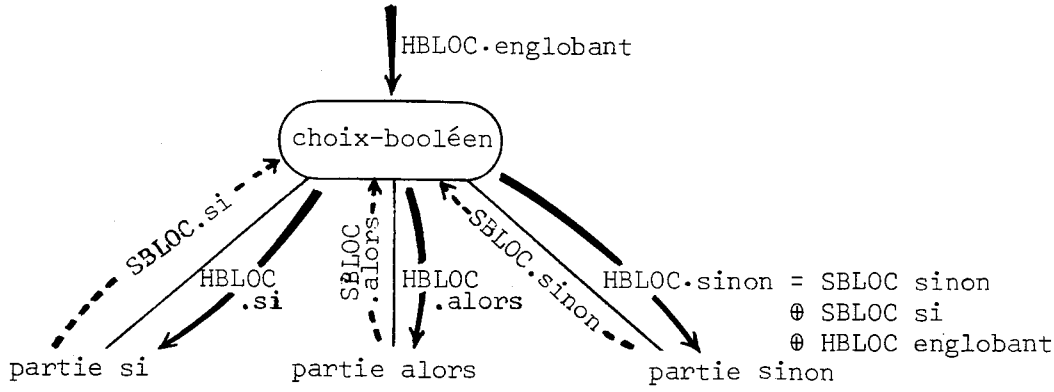
La construction d'un attribut HBLOC est faite au niveau d'un noeud délimitant une région, en fonction des attributs des noeuds adjacents. Pour le noeud (début), la règle est la suivante :



HBLOC.série ← SBLOC.série ⊕ HBLOC.début

Elle traduit l'imbrication des régions. Lors de la recherche d'un identificateur, l'ensemble des déclarations propres à la région (SBLOC.série) est exploré avant l'ensemble des déclarations des régions englobantes. L'identification d'un objet dans HBLOC.série se fait en recherchant d'abord cet objet dans SBLOC.série, puis dans HBLOC.début, afin de respecter la règle de redéclaration d'un identificateur dans un bloc interne.

Dans le cas des propositions conditionnelles, entre autres, la règle est différente, puisque les déclarations faites dans la *partie si* sont accessibles dans la *partie alors* et la *partie sinon*, ce qui se traduit par :



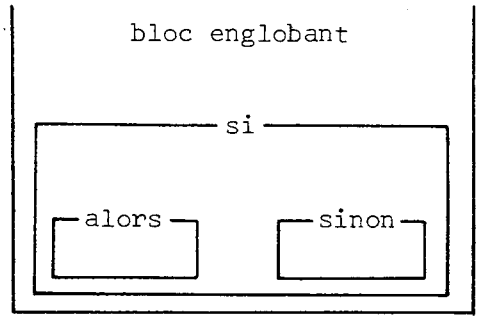
Les règles d'attributs sont :

HBLOC.partie si ←
  SBLOC.partie si co les déclarations de la *partie si* co
⊕ HBLOC.choix-booléen co les déclarations de la région englobante co

HBLOC.partie alors ←
  SBLOC.partie alors co les déclarations de la *partie alors* co
⊕ SBLOC.partie si co les déclarations de la *partie si* co
⊕ HBLOC.choix-booléen co les déclarations de la région englobante co

HBLOC.partie sinon ←
  SBLOC.partie sinon co les déclarations de la *partie sinon* co
⊕ SBLOC.partie si co les déclarations de la *partie si* co
⊕ HBLOC.choix-booléen co les déclarations de la région englobante co

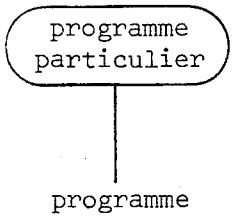
Ce qui correspond au schéma d'imbrication ci-dessus :



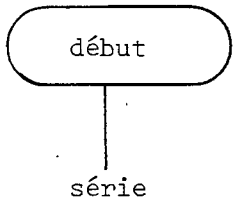
Nous n'avons pas précisé la nature des attributs HBLOC et SBLOC. Nous le faisons dans la troisième partie, lorsque nous montrons une implantation particulière. Pour l'instant, nous les considérons comme des ensembles d'éléments du type identificateur et opérateur. L'opérateur + représente l'union ordonnée d'ensembles (concaténation). Lors de la recherche d'un objet dans S1 + S2, l'ensemble S1 sera exploré avant S2, ce qui permet de traduire l'imbrication des blocs.

Nous donnons maintenant le système d'attributs sur l'arbre abstrait, qui décrit le mécanisme d'identification des identificateurs.

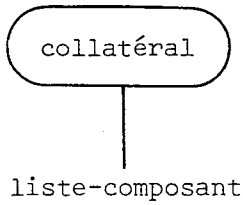
2 - LE SYSTEME D'ATTRIBUTS -



HBLOC.programme ← environnement standard
co L'environnement standard est l'ensemble des opérateurs et identificateurs fournis à tout programme
co



HBLOC.série ← SBLOC.série + HBLOC.début
co Aux déclarations de la série, qui sont synthétisées, s'ajoutent celles de la région englobante
co



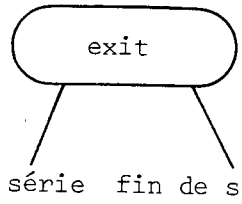
(HBLOC.composant ← HBLOC.collatéral)*
co Les unités composant une proposition collatérale ne contiennent pas de déclaration. L'environnement englobant est hérité sur chaque composant.
co

SBLOC.série ← {SBLOC.composant/
composant ∈ listecomposants }

co l'attribut SBLOC d'une série est constitué de l'ensemble des valeurs de type identificateur et opérateur synthétisé par chaque déclaration de la liste de déclarations et d'unités qui précède la dernière unité. l'attribut HBLOC qui est hérité sur chacun des constituants de la série est calculé au niveau du noeud père de (série). co

(HBLOC.unité ← HBLOC.série)*

HBLOC.dernière unité ← HBLOC.série

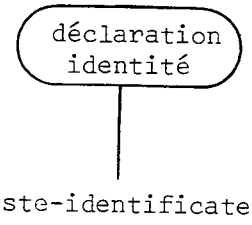


SBLOC.exit ← SBLOC.série

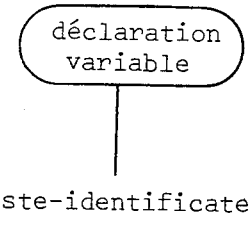
co cette synthèse n'a de sens que pour le premier *exit* d'une série, car après on ne peut plus trouver de déclaration co

HBLOC.série ← HBLOC.exit

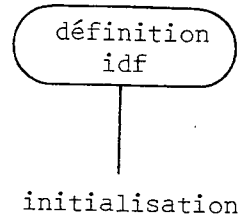
HBLOC.fin de série ← HBLOC.exit



SBLOC.déclaration identité ←
{SBLOC.identificateur/
identificateur ∈ liste-identificateurs}



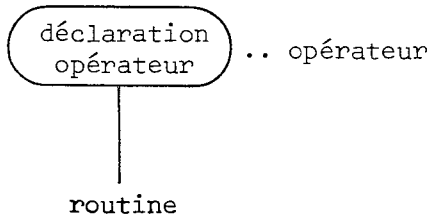
SBLOC.déclaration variable ←
{SBLOC.identificateur/
identificateur ∈ liste-identificateurs}



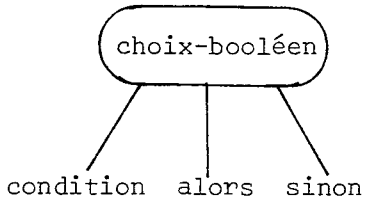
SBLOC.définition idf ←
identificateur de (définition idf)

HBLOC.initialisation ←
HBLOC.définition idf

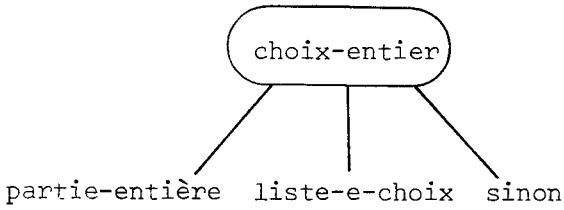
co création effective de SBLOC au niveau d'une déclaration co



SBLOC.déclaration opérateur ←
opérateur de (déclaration opérateur)
HBLOC.routine ←
HBLOC.déclaration opérateur
co création effective de SBLOC au niveau d'une déclaration co

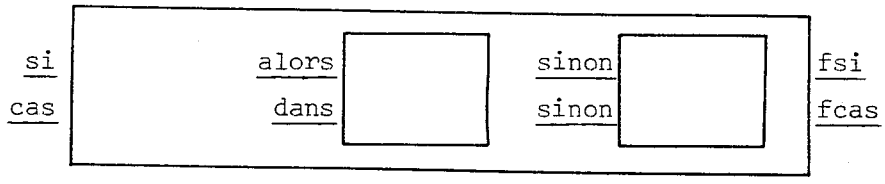


HBLOC.condition ← SBLOC.condition +
HBLOC.choix-booléen
HBLOC.alors ← SBLOC.alors + SBLOC.condition +
HBLOC.choix-booléen
HBLOC.sinon ← SBLOC.sinon + SBLOC.condition +
HBLOC.choix-booléen

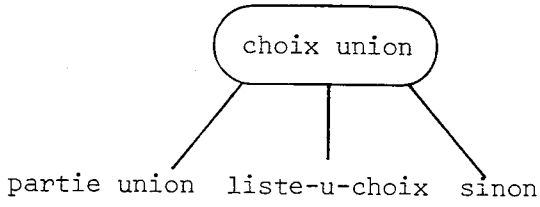


HBLOC.partie entière ← SBLOC.partie entière +
HBLOC.choix entier
(HBLOC.e-choix ← SBLOC.partie entière
+ HBLOC.choix entier)*
HBLOC.sinon ← SBLOC.sinon +
SBLOC.partie entière +
HBLOC.choix entier

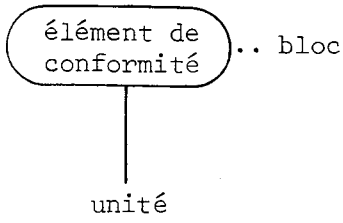
co schéma d'imbrication des régions dans les propositions choix



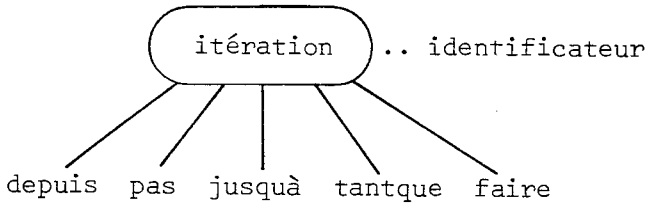
co .




```
HBLOC.partie union ← SBLOC.partie union +
                     HBLOC.choix union
(HBLOC.u-choix ← SBLOC.partie union
                 + HBLOC.choix union)*
HBLOC.sinon ← SBLOC.sinon +
              SBLOC.partie union +
              HBLOC.choix union
```

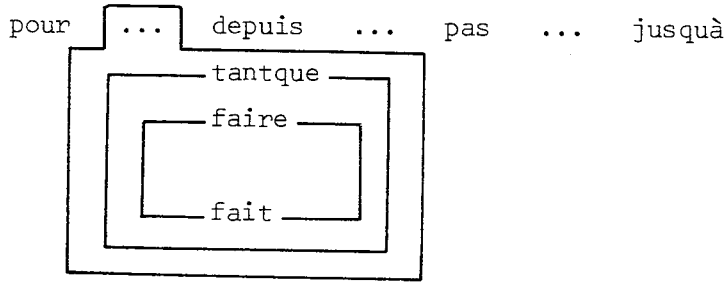



```
HBLOC.unité ← bloc de (élement de conformit
              + HBLOC.élément de conformité
```

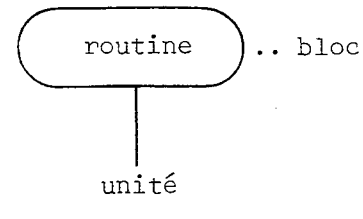



```
HBLOC.depuis ← HBLOC.itération
HBLOC.pas    ← HBLOC.itération
HBLOC.jusquà ← HBLOC.itération
HBLOC.tantque← SBLOC.tantque + identificateur de (itération)
                               + HBLOC.itération
HBLOC.faire  ← SBLOC.faire + SBLOC.tantque
                           + identificateur de (itération)
                           + HBLOC.itération
```

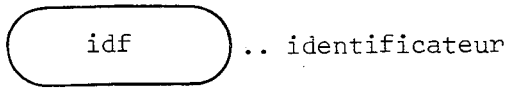
co schéma d'imbrication des régions



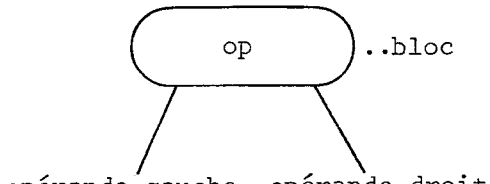
co



HBLOC.unité ← bloc de (routine) +
HBLOC.routine



identificateur de (idf) ←
identifier (no idf de (idf),
HBLOC.idf)



bloc de (op) ← HBLOC.op
co le noeud est décoré par l'ensemble des déclarations accessibles à ce point, pour permettre l'identification de l'opérateur lors du traitement des modifications co
HBLOC.opérande gauche ← HBLOC.op
HBLOC.opérande droit ← HBLOC.op

3 - REMARQUES -

Les noeuds qui n'apparaissent pas au paragraphe précédent sont simplement transmetteurs.

La deuxième phase de décoration, le traitement des modifications,va utiliser les informations qui viennent d'être mises en place. Ces deux phases pourraient être regroupées. Nous les avons séparées pour ne pas alourdir la description. Cette séparation conduit à s'écarter de certains aspects des attributs. En effet, certains attributs utilisés lors de la seconde phase "dépendent" au - au sens donné par Lorho en [LORHO] - d'attributs de la première phase. Nous avons considéré ces derniers comme des "décorations" de l'arbre abstrait. Or, cette notion de décoration peut être considérée de deux points de vue. Dans un système d'attributs l'arbre est décoré lorsque ses attributs sont évalués, et chaque attribut constitue une décoration. Pour l'écrivain de compilateurs une décoration peut être une valeur mise en place au cours d'une phase de traitement et utilisée ultérieurement. Notre point de vue est mixte. Les décorations que nous utilisons sont effectivement des attributs, mais nous n'en considérons qu'une partie, et elles servent à transmettre de l'information d'une phase à l'autre.

II.3.4 - Deuxième phase de décoration : traitement des modifications -

1 - PRINCIPE -

En Algol 68 tout objet du programme a un mode propre, ou mode a priori. Ainsi les notations *vrai*, *faux* ont pour mode a priori le mode *bool*, la notation *5* le mode *ent*, l'identificateur *x* le mode correspondant à sa déclaration, par exemple *rep réel*. Une construction comme l'affectation *x := 5* a pour mode a priori celui de sa partie gauche, ici *rep réel*.

A toute construction du langage sont également associés une force de contexte et un mode requis au mode a posteriori. Le mode a priori d'une valeur n'est pas toujours celui qui convient, ce dernier étant précisément le mode a posteriori de la valeur. Il faut donc émettre les conversions ou modifications qui transforment le mode a priori en mode a posteriori. Les modifications possibles dépendent de la force de contexte de l'objet.

Exemple : *x := 5*

La force de contexte de la source *5* est *fort*
Son mode a priori est *ent*
Son mode a posteriori est *réel*
La modification émise est l'*élargissement* d'entier à réel

La même modification, élargissement, ne serait pas permise en opérande de formule, où la force de contexte est *ferme*, car le même opérateur peut être défini entre opérandes de mode entier et réel. Dans cette situation, la possibilité d'élargissement introduirait une ambiguïté.

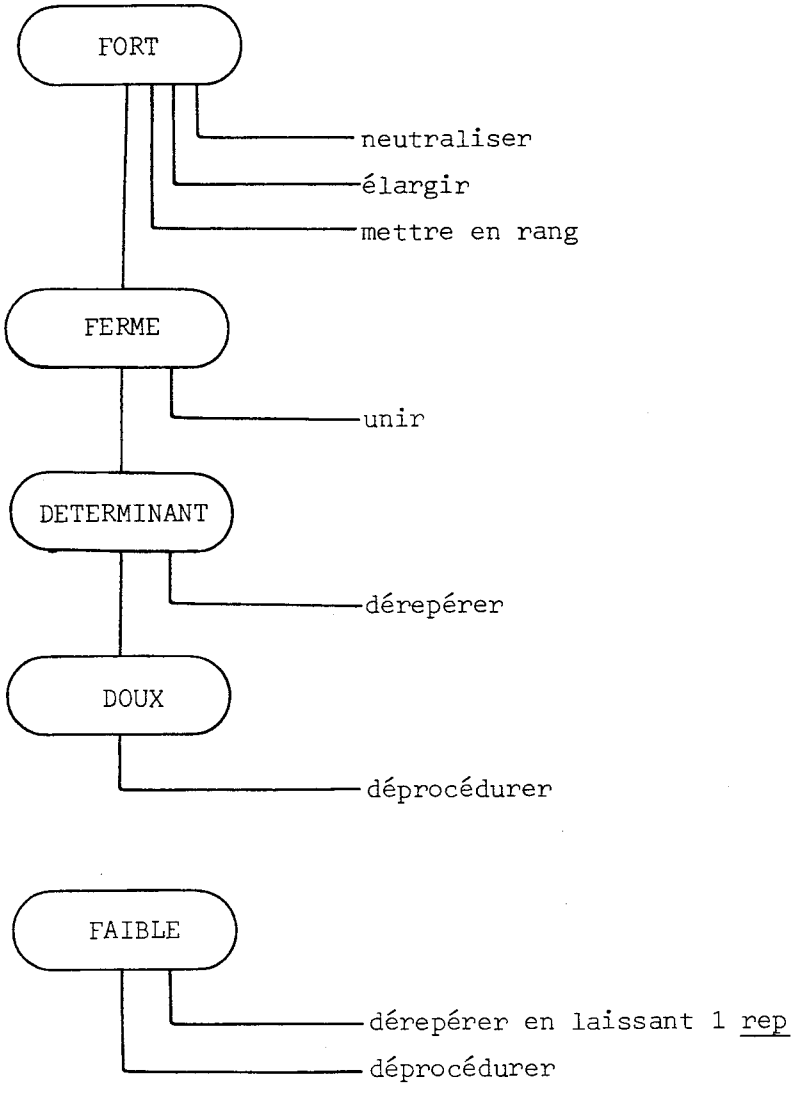
En effet, considérons les déclarations d'opérateurs :

op (ent, ent) p = ...
op (réel, réel) p = ...

et la formule *i + 1* où *i* est entier. Si la modification d'entier à réel est possible, on ne sait pas quel opérateur *p* choisir.

Les règles qui précisent la détermination de la force de contexte, du mode a posteriori et les modifications à émettre, sont attachées à chaque construction syntaxique. Elles sont exprimées dans le Rapport sous la forme d'une W-grammaire. Nous les décrivons à l'aide d'un système d'attributs sur l'arbre abstrait.

La force de contexte est représentée par l'attribut hérité FC. Sa valeur est déterminée au niveau de certains noeuds et simplement transmise par d'autres. On peut considérer cinq forces de contexte : *fort*, *ferme*, *déterminant*, *faible* et *doux*, (les termes originaux sont : *strong*, *firm*, *meek*, *weak* et *soft*). Les modifications possibles avec ces forces de contexte sont décrites par le diagramme ci-dessous, qui montre l'inclusion de DOUX dans DETERMINANT et ainsi de suite.



Ainsi la force de contexte en partie gauche de l'affectation a la valeur *doux* et en partie droite la valeur *fort*. La partie gauche pourra dont être seulement *déprocédurée*. Par contre, en partie droite, toutes les modifications sont permises pour arriver au mode demandé. A la racine de l'arbre, la valeur de la force de contexte est *fort*.

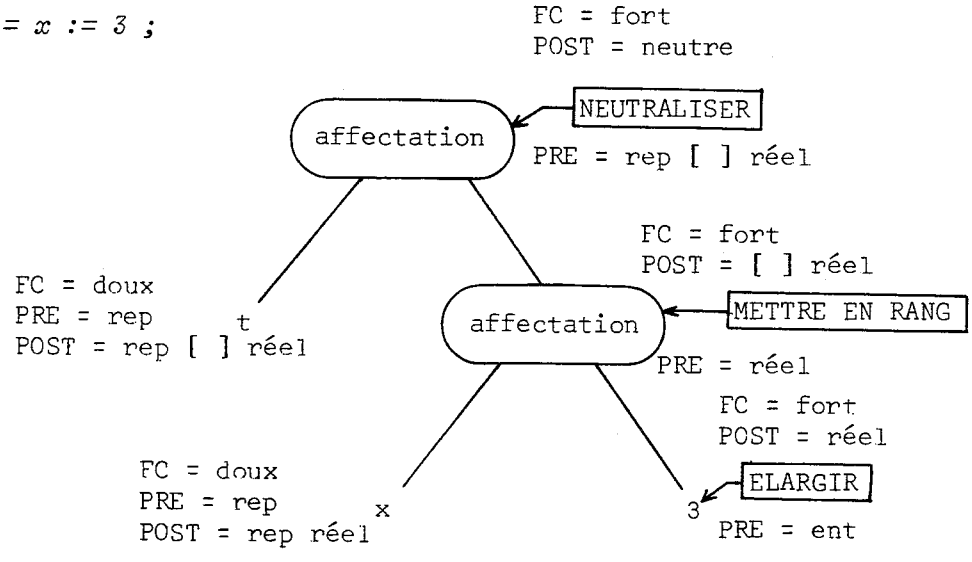
Le mode a posteriori est représenté par l'attribut hérité POST. La valeur de ce dernier est déterminée au niveau de certains noeuds, éventuellement en fonction d'autres attributs, et transmise par d'autres noeuds. Il est parfois incomplet. Ainsi, en partie gauche d'affectation, avec la force de contexte *doux*, le mode a posteriori est constitué du seul préfixe rep. Dans ce cas, le mode a priori synthétisé constitue justement le mode complet.

Le mode a priori est représenté par l'attribut synthétisé PRE. Sa valeur est déterminée au niveau des feuilles (mode d'un identificateur, mode d'une notation) ou au niveau de certains noeuds intermédiaires (résultat d'un opérateur).

Au niveau de certains noeuds (coercends) le mode a priori et le mode a posterior sont examinés et il faut trouver une transformation du premier au second, utilisant les modifications permises par la force de contexte courante. Ce problème est résolu par une fonction MODIFIER qui a pour paramètres FC, PRE et POST. Son rôle est de trouver un chemin entre PRE et POST en émettant la liste de modifications correspondantes, qui va constituer une décoration du noeud considéré. Le résultat de cette fonction est une valeur constituée de deux éléments. Le premier élément est le mode a priori qui va être remonté ; ce mode peut être le *mode erreur* si aucun chemin n'a été trouvé entre PRE et POST. Le second élément est la liste de modifications créée par la fonction MODIFIER, qui va constituer une décoration du noeud.

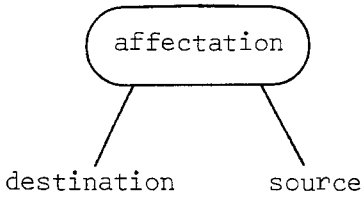
Nous détaillons maintenant le traitement des modifications sur un exemple. Nous donnons d'abord l'arbre avec tous les attributs évaluées, puis les règles d'évaluation et enfin les étapes de l'évaluation.

Exemple : *flex*[*1 : n*] *réel t ; réel x ;*
t := x := 3 ;



La figure montre les valeurs des attributs force de contexte prémode et postmode ainsi que les décorations du type modification sur l'arbre abstrait correspondant à l'exemple ci-dessus.

Ce résultat est obtenu par l'élaboration d'un système d'attributs sur l'arbre abstrait. Les règles correspondant à l'affectation sont :

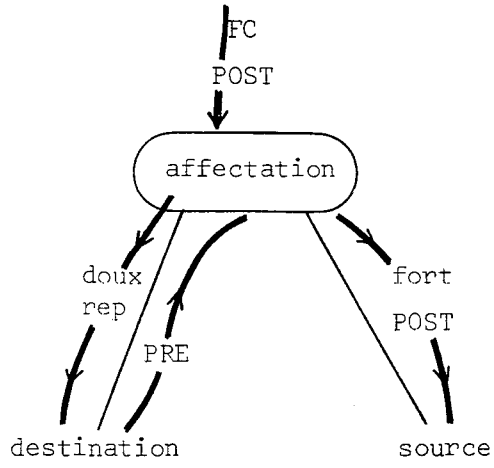


FC.destination ← doux
POST.destination ← rep

FC.source ← fort
POST.source ← dérep(PRE.destination)

PRE.*affectation* ← modifier(FC.*affectation*
POST.*affectation*
PRE.destination)

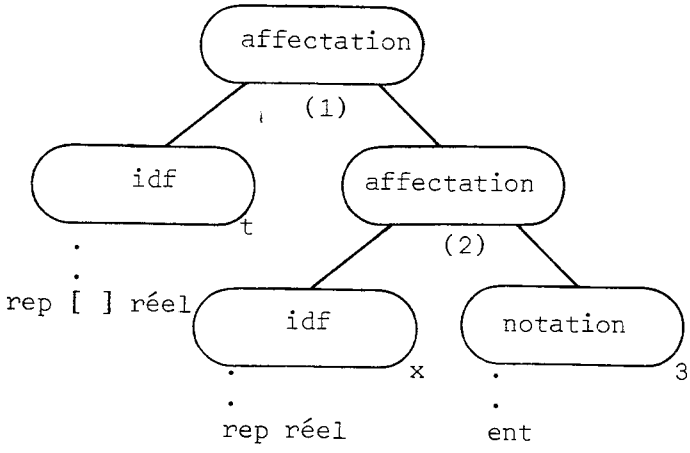
Voyons au niveau de ce noeud quel peut être le schéma d'évaluation des attributs. Nous avons regroupé les règles dans un ordre qui peut constituer un ordre d'évaluation, mais cet ordre n'est pas unique,et l'ordre proposé n'a rien d'impératif. En effet, seul le graphe de dépendance des attributs impose des contraintes d'évaluation. On peut proposer le schéma suivant :



le lecteur peut avoir l'impression d'une certaine complexité. Nous rappelons qu'en général le programmeur n'a pas à se soucier de ces mécanismes. Ils sont utiles pour une définition précise du langage et pour son implantation. Ils assurent la cohérence du traitement des modes et des conversions, les règles d'utilisation demeurant simples et naturelles. De plus, il n'y a pas nécessairement de génération de code pour les modifications.

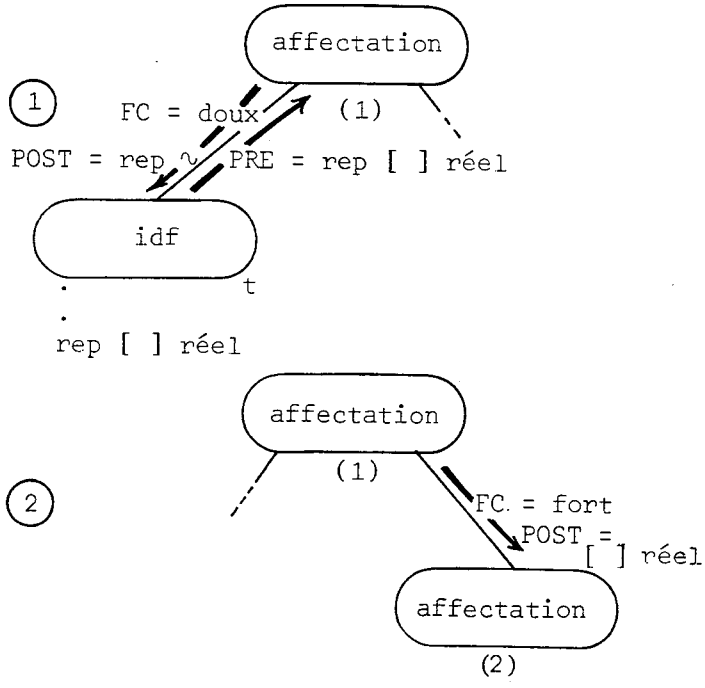
application à l'exemple : *t := x := 3 ;*

L'arbre de départ est le suivant :



Les identificateurs sont déjà identifiés et leur mode est attaché aux noeuds (idf) de l'arbre abstrait. Il va constituer l'attribut PRE de ces noeuds.

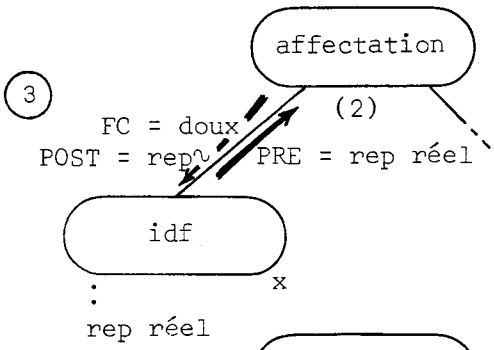
L'évaluation des attributs de ce sous-arbre se fait indépendamment de la force de contexte et du mode a posteriori hérités. Ici FC.*affectation* = fort et POST.*affectation* = neutre.



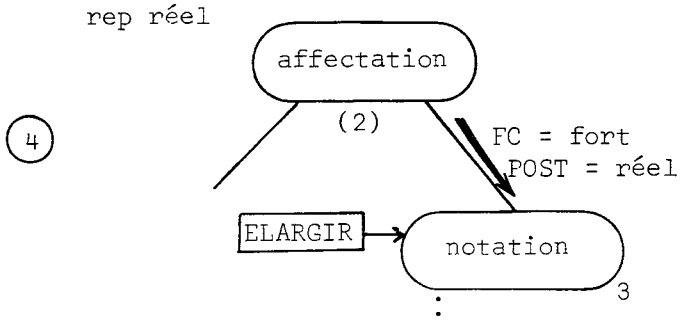
évaluation des attributs du fils gauche. Le mode a priori remonté : rep [ ] réel est utilisé pour déterminer le mode a posteriori de la partie droite (cf. (2)) et pour construire la liste des mofications attachées au noeud (cf (6)).

évaluation des attributs du fils droit. Le travail sur le sous-arbre de *affectation* est indépendant du mode a posteriori demandé, ici [ ]ré

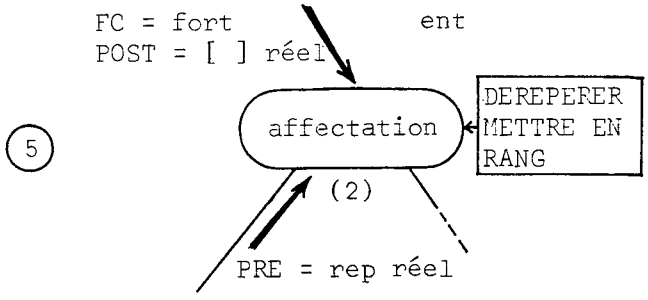
Lorsque l'évaluation des attributs de ce sous-arbre est terminé on peut appliquer la fonction *modifier* au niveau de ce noeud - cf (5) -.



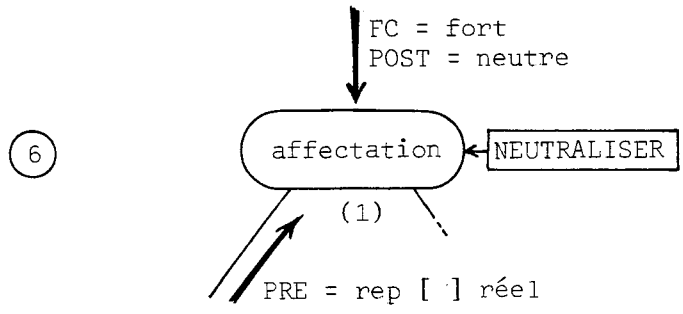
évaluation des attributs du fils gauche. Analogue à (1) .



évaluation des attributs du fils droit. La fonction *modifier* est appliquée à la notation 3 et conduit à émettre la modification *élargissement* d'entier à réel.

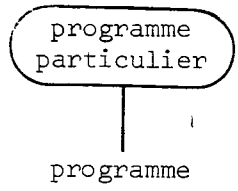


Application de la fonction *modifier*. Emission de la modification *dérepérer* qui transforme le mode a priori rep réel en réel, puis de la modification *mettre en rang*, qui transforme le mode réel en tableau de réel, [ ] réel

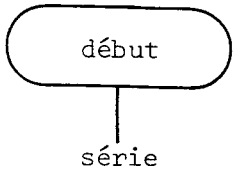


Application de la fonction *modifier*. Cette construction fait partie d'une proposition sérielle. Elle est *neutralisée*, c'est-à-dire que sa valeur n'est pas considérée.

2 - LE SYSTEME D'ATTRIBUTS -

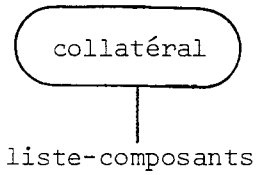


FC. programme ← fort
POST.programme ← neutre

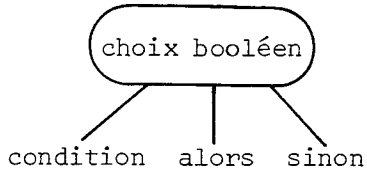


FC. série ← FC. début
POST.série ← POST.début
PRE. début ← PRE. série

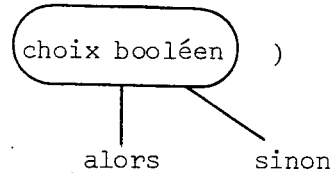
co Les attributs sont transmis co




```
si    FC.collatéral = fort
alors cas type POST.collatéral
      dans co le mode a priori du noeud est inconnu. Il dépend du
              contexte d'utilisation
           co
  neutre : co descendre (fort, neutre) sur chaque unité de la liste des
              composants
           co
              (FC.  composant ← fort  )*
              (POST.composant ← neutre)*

  struct : co le mode requis est un mode de type struct (   ).
              Chaque unité correspond à un champ de la structure.
           co
              (FC.  composant  ← fort)*
              (POST.composant_i ← mode champ (POST.collatéral, i))*

  rang   : co le mode requis est du type [   ] m.
              chaque unité doit être du type m.
           co
              (FC.  composant ← fort)*
              (POST.composant ← mode élément (POST.collatéral))*
      sinon   erreur
      fincas
sinon erreur
finsi
```

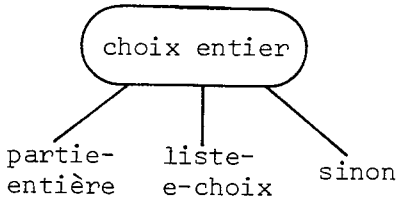


FC. condition ← déterminant
POST.condition ← bool

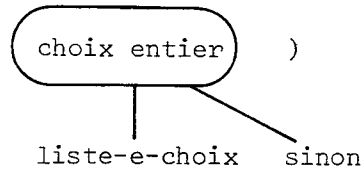
équilibrage (

)

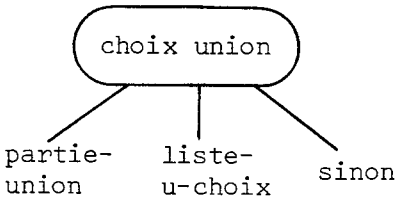
co le noeud est passé en paramètre
avec ses attributs
co



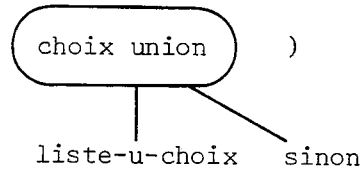
FC. partie-entière ← déterminant
POST.partie-entière ← ent

équilibrage (

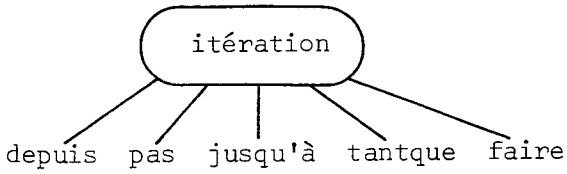
)



FC. partie-union ← déterminant
POST.partie-union ← union

équilibrage (

)



FC. depuis ⟵ déterminant
POST.depuis ⟵ ent

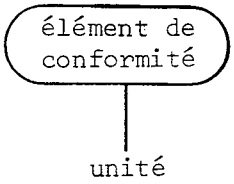
FC. pas ⟵ déterminant
POST.pas ⟵ ent

FC. jusqu'à ⟵ déterminant
POST.jusqu'à ⟵ ent

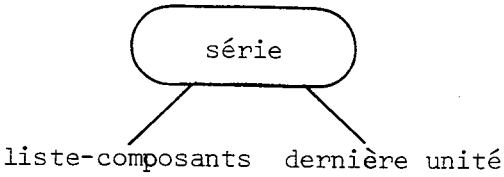
FC. tantque ⟵ déterminant
POST.tantque ⟵ bool

FC. faire ⟵ fort
POST.faire ⟵ neutre

PRE.itération ⟵neutre



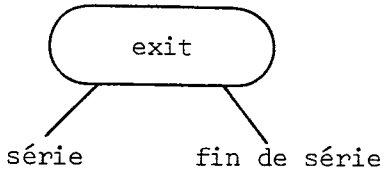
FC. unité ⟵ FC. élément de conformité
POST.unité ⟵ POST.élément de conformité
PRE.élément de conformité ⟵ PRE.unité

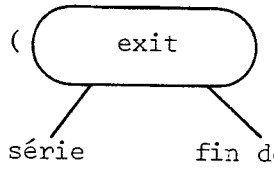


(FC. composant ← fort)*
(POST.composant ← neutre)*

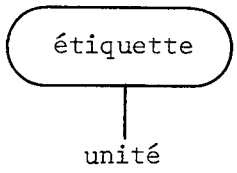
co la dernière unité est "significative" (voir ch.III.3) c'est-à-dire que son mode est celui de la série co

FC. dernière unité ← FC. série
POST.dernière unité ← POST.série
PRE.série ← PRE.dernière unité

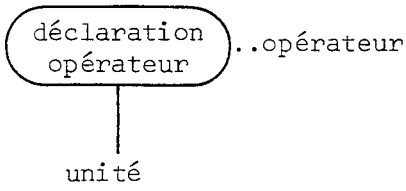


balance ()

co équilibrage co



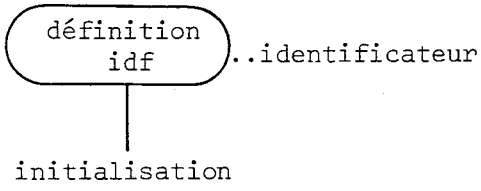
co attributs transmis co
FC. unité ← FC. étiquette
POST.unité ← POST.étiquette
PRE.étiquette ← PRE.unité



FC. unité ← fort

POST.unité ← mode résultat

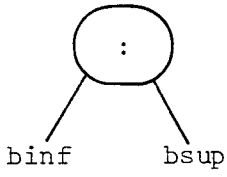
(opérateur de déclaration opérateur)



FC. initialisation ← fort

POST.initialisation ← mode identificateur

(identificateur de définition idf)



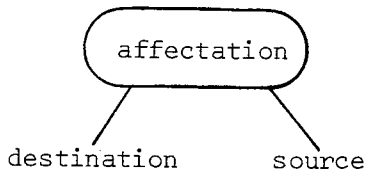
co bornes dans un déclareur de tableau co

FC. binf ← déterminant

FC. bsup ← déterminant

POST.binf ← ent

POST.bsup ← ent

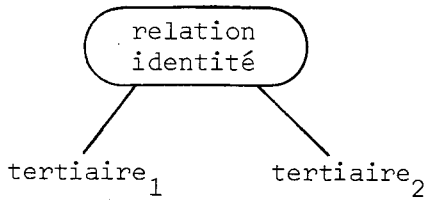


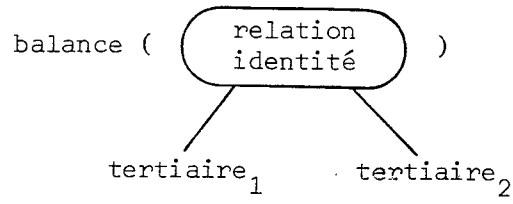
FC. destination ← doux
POST.destination ← rep

co le mode a posteriori complet est inconnu. Seul son préfixe : rep est déterminé. Le mode a priori synthétisé constitue le mode complet. Il sert à calculer le mode a posteriori de la source
co

FC. source ← fort
POST.source ← sans rep (PRE.destination)

(PRE.affectation, modification de (affectation)) ←
modifier (FC.affectation, POST.affectation, PRE.destination)
co le mode a priori de l'affectation est celui de la destination co





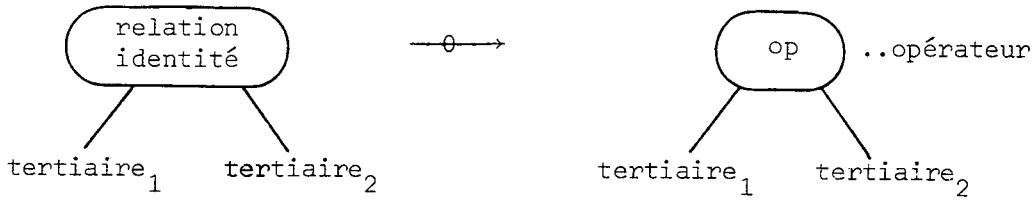
(PRE.relation identité, modification de (relation identité)) ←-

modifier (FC.relation identité, POST.relation identité, bool)

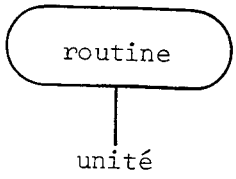
co le noeud (relation identité) est transformé en noeud (op)

pour la génération de code, avec la même décoration modification et une décoration opérateur particulière.

co



opérateur de (op) ← relation identité.

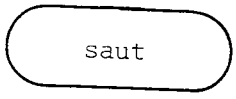


FC. unité ← fort

POST.unité ← mode résultat

(mode de routine)

PRE.routine ← mode de routine

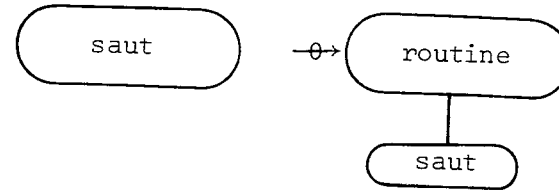


si FC.saut = fort

alors mode de saut ← POST.saut

si préfixe (POST.saut) = proc

alors co transformer en un noeud routine pour la génération co



fsi

PRE.saut ← POST.saut

sinon PRE.saut ← mode erreur

co un saut peut apparaître seulement avec la force de contexte "fort", auquel cas il peut être modifié à tout mode co

finsi

mode de saut ← PRE.saut

fant

```
co peut être modifié à tout mode avec la
   force de contexte "fort"
co
si    FC.fant = fort
alors PRE.fant ← POST.fant
sinon PRE.fant ← mode erreur
finsi
mode de ( fant ) ← PRE.fant
```

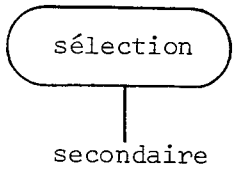
nil

```
co peut être modifié à tout mode de type
   repère, avec la force de contexte "fort"
co
si    FC.nil = fort et
      préfixe (POST.nil) = rep
alors PRE.nil ← POST.nil
sinon PRE.nil ← mode erreur
finsi
mode de ( nil ) ← PRE.nil
```

générateur

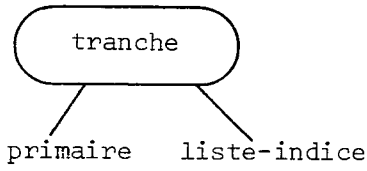
```
co s-générateur
   d-générateur
   i-générateur
co
```

(PRE.générateur, modification de (générateur)) ←
modifier (FC.générateur, POST.générateur, mode de (générateur))



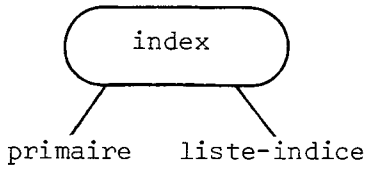
FC. secondaire ← faible
POST.secondaire ← struct

(PRE.sélection, modification de (sélection)
modifier (FC. sélection, POST.sélection,
mode champ (PRE.secondaire,
champ de (sélection)))



FC. primaire ← faible
POST.primaire ← rang

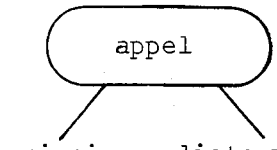
(PRE.tranche, modification de (tranche))
modifier (FC. tranche, POST.tranche,
mode tranche (PRE.primaire, liste indice))
(FC. indice ← déterminant)*
(POST.indice ← ent)*



FC. primaire ← faible
POST.primaire ← rang

(PRE.index, modification de (index))←
modifier (FC.index, POST.index,
mode élément (PRE.primaire))

(FC. indice ← déterminant)*
(POST.indice ← ent)*



FC. primaire ← faible
POST.primaire ← proc

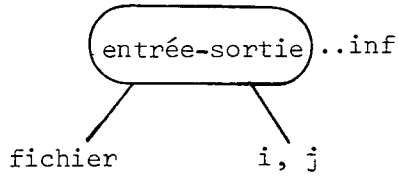
(PRE.appel, modification de (appel))←
modifier (FC.appel, POST.appel,
mode résultat (PRE.primaire))

(FC. paramètre ← fort)*
(POST.paramètre$_i$ ← mode paramètre
(PRE.primaire, i))*

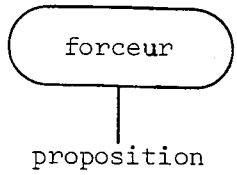
co lorsque le primaire de l'appel spécifie une procédure d'entrée-sortie, le noeud est transformé en noeud

où "fichier" est le premier paramètre de l'appel. La décoration type provient du primaire.

L'appel *get ((fichier, i, j))* donne l'arbre suivant :



co



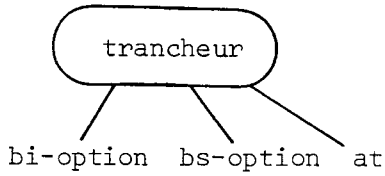
FC. proposition ← fort

POST.proposition ← mode de (forceur)

(PRE.forceur, modification de (forceur)) ←

modifier (FC. forceur, POST.forceur,

mode de (forceur))



FC. bi-option ← déterminant

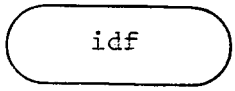
POST.bi-option ← ent

FC. bs-option ← déterminant

POST.bs-option ← ent

FC. at ← déterminant

POST.at ← ent



(PRE.idf, modification de (idf)) ←

modifier (FC.idf, POST.idf, mode de (idf))

L'aspect déclaratif du système d'attributs a été conservé dans cette description. Il reste deux points à traiter : l'équilibrage et l'identification des opérateurs. Nous avons rencontré quelques problèmes pour utiliser le même outil de description. C'est l'objet du prochain paragraphe.

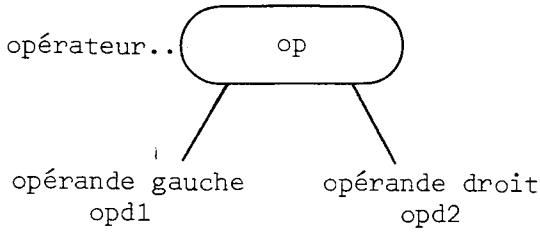
vacuum

si FC.vacuum = fort et
préfixe (POST.vacuum) = rang
alors PRE.vacuum ← POST.vacuum
sinon PRE.vacuum ← mode erreur
finsi
mode de (vacuum) ← PRE.vacuum

notation

(PRE.notation, modification de (notation)) ←
modifier (FC.notation, POST.notation,
mode de (notation))

3 - IDENTFICATION DES OPERATEURS -



co Lors de la construction de l'arbre abstrait, le noeud (op) est décoré par le numéro de l'opérateur, qui va servir a le retrouver dans l'ensemble des déclarations (*bloc*) accessibles à ce point. *Bloc* est également une décoration du noeud, mise en place par la phase de décoration précédente.

Chaque opérateur a un mode du type proc(m1, m2) m3 où m1 et m2 sont les modes des opérandes et m3 le mode du résultat. Le principe de l'identification consiste à rechercher dans l'ensemble des opérateurs de la région, celui dont les modes m1 et m2 sont modifiables à ceux des opérandes gauche et droit avec la force de contexte *ferme*. Pour cela il suffit d'évaluer les attributs de sous-arbres des opérandes avec m1 et m2 comme mode a priori. Si le mode a posteriori remonté n'est pas le mode erreur, c'est que l'opérateur convient. Sinon, il faut essayer un autre opérateur, ce qui conduit à une nouvelle évaluation des attributs des sous-arbres des opérandes, et ainsi de suite jusqu'à trouver le bon opérateur.

Pour cela nous introduisons explicitement une fonction d'évaluation des attributs d'un sous-arbre en fonction des attributs hérités du noeud père.

Exemple : FC.opérande gauche ← ferme
POST.opérande gauche ← m1
éval(opérande gauche)

Si une nouvelle valeur est donnée à POST.opérande gauche alors "éval" sera de nouveau activé et conduira au calcul d'une nouvelle valeur de PRE.opérande gauche.
co

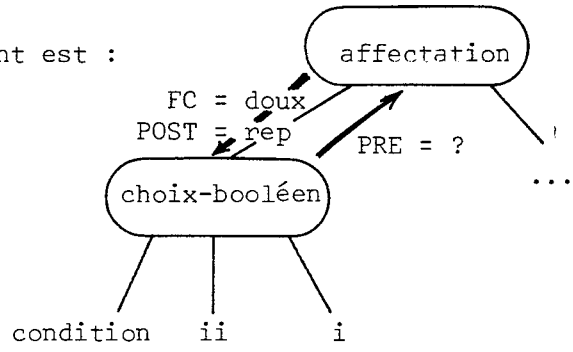
```
bool trouve ← faux
pour tout opérateur q ∈ bloc de (op)
tant que ¬trouve
faire    m1 ← mode opérande gauche (q)
         m2 ← mode opérande droit (q)
         FC.opérande gauche ← ferme
         POST.opérande gauche ← m1
         éval (opérande gauche)
         si PRE.opérande gauche ≠ mode erreur
            et opérande droit ≠/\ co opérateur unaire co
         alors FC.opérande droit ← ferme
               POST.opérande droit ← m2
               éval (opérande droit)
               si PRE.opérande droit ≠ mode erreur
               alors trouve ← vrai
               finsi
         finsi
finfaire

(PRE.op, modification de (op) ) ←
modifier (FC.op, POST.op, mode résultat (q))
```

4 - EQUILIBRAGE -

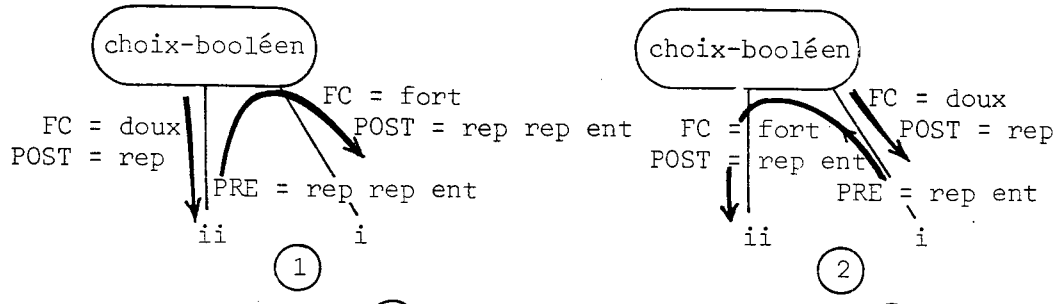
L'équilibrage est une qualité des couples d'opérandes de certains noeuds, qui permet de considérer l'un des opérandes avec la force de contexte fort, quelle que soit la force de contexte du noeud.

Exemple : *rep ent ii ; co ii est de mode rep rep ent co*
ent i ; co i est de mode rep ent co
si condition alors ii sinon i fsi := ...

L'arbre abstrait correspondant est :

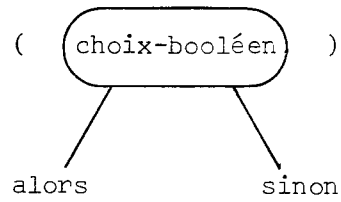


Il s'agit de déterminer le mode a priori de la proposition choix. Celui-ci sera rep réel ou rep rep réel, qui sont les modes des opérandes. La force de contexte du noeud père est *doux*. L'un des opérandes (alors ou sinon) doit avoir la force de contexte *doux* avec le mode a posteriori rep et l'autre la force de contexte "fort" avec pour mode a posteriori le mode a priori de l'autre opérande. On a donc l'un des deux schémas suivants :



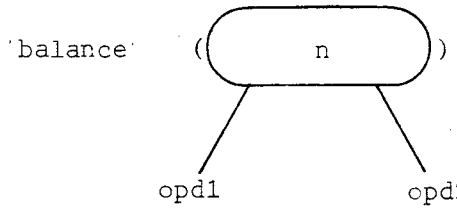
On voit que seul le schéma (2) convient car dans le schéma (1), i ne peut être modifié au mode rep rep ent tandis que dans le schéma (2) ii peut être modifié au mode rep ent.

Pour le traitement de l'équilibrage nous avons défini une fonction balance qui s'applique au noeud et à ses deux opérandes en position d'équilibrage. Les paramètres sont donc le noeud - avec ses attributs et décorations - et les deux opérandes.

Exemple : balance (choix-booléen) 

Son principe est le suivant :

évaluer les attributs de l'un des opérandes avec la force de contexte héritée. Si un mode a priori est retourné, essayer le second opérande avec ce mode pour mode a posteriori et la force de contexte fort. Si le mode a priori retourné n'est pas le mode erreur, l'évaluation est terminée. Sinon il faut interchanger les opérandes, ce qui conduit à une deuxième évaluation des attributs des opérandes.



```
si FC.n = fort
alors co pas d'équilibrage co
     FC.opd1    ← fort
     FC.opd2    ← fort
     POST.opd1  ← POST.n
     POST.opd2  ← POST.n
     éval(opd1)
     éval(opd2)
sinon co équilibrage : essayer un opérande avec les attributs FC et POST
         du noeud (n)
      co
     FC.opd1   ← FC.n
     POST.opd1 ← POST.n
     éval(opd1)
     si PRE.opd1 ≠ mode erreur
     alors co essayer l'autre opérande avec la force de contexte "fort" et
              le mode a priori du premier opérande si le mode a priori du
              noeud est incomplet :
             -union avec la force de contexte déterminant
             -rep avec la force de contexte doux
             -proc ou rang avec la force de contexte faible
           co
           FC.opd2   ← fort
           POST.opd2 ← si complet (POST.n)
                       alors POST.n
                       sinon PRE.opd1
                       finsi
           éval (opd2)
           PRE.n ← PRE.opd2
     sinon co essai du second opérande avec les attributs FC et POST du noeud (n)
           co
           FC.opd2   ← FC.n
           POST.opd2 ← POST.n
           éval (opd2)
```

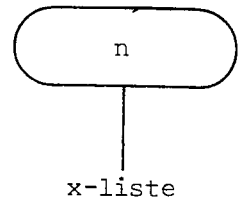
```
            si PRE.opd2 ≠ mode erreur
            alors co essayer de nouveau le premier opérande, mais cette fois
                    avec la force de contexte "fort"
                 co
                 FC.opd1   ← fort
                 POST.opd1 ← si complet (POST.n)
                             alors POST.n
                             sinon PRE.opd2
                             finsi
                 éval (opd1)
                 PRE.n ← PRE.opd1
            sinon PRE.n ← mode erreur
            finsi
      finsi
finsi
```

Les cas d'équilibrage sont les suivants :

- choix-booléen : équilibrage entre "partie alors" et "partie sinon"
- choix-entier : équilibrage entre "partie alors" et "partie sinon"
- choix-union : équilibrage entre "partie alors" et "partie sinon"
- exit : équilibrage entre "série" et "fin de série"
- relation d'identité : équilibrage entre les deux opérandes
- liste d'unités de choix-entier
- liste d'éléments de conformité de choix-union

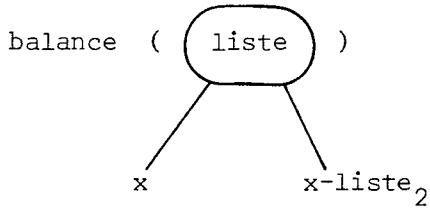
Dans ces deux derniers cas, les règles n'ont pas été explicitées. Nous en donnons ici le principe. Il s'agit de noeuds dont les fils forment une liste, donc de la forme :

n est *choix-entier* ou *choix-union*
x est unité ou élément de conformité



L'équilibrage sur l'arbre x-liste peut être décrit par :

si x-liste s'écrit x co la liste contient un seul élément co
alors éval (x)
sinon x-liste s'écrit x ⊕ x-liste 2
co il y a au moins deux fils, donc équilibrage co



finsi

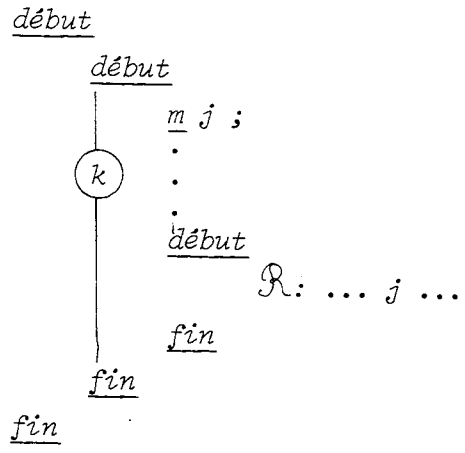
5 - PORTEE DES ROUTINES -

Le calcul de la portée des routines nécessite la connaissance de la portée des identificateurs et des opérateurs, ce qui suppose réalisée leur identification, c'est pourquoi nous le présentons maintenant.

Selon la définition donnée dans le Rapport au paragraphe 2.1.3.5. Une routine est un objet, composé d'un *texte de routine* T et d'un environ E. Cet environ est défini comme étant celui nécessaire à T dans l'environ de définition E1. (Rapport : § 5.4.1.2) et (Rapport : 7.2.2.c). La portée de la routine est celle de cet environ E.

La portée de cet environ E est la plus restreinte parmi les portées des objets (indicateur de mode avec déclareur actuel, opérateur, identificateur) utilisés par cette routine.

La raison de cette limitation est que la routine doit pouvoir accéder à ces objets ou aux valeurs qu'ils repèrent, au moment de son exécution. Lors de l'appel de la routine, ces objets doivent donc être présents dans la mémoire (pile ou tas). Considérons le programme suivant :

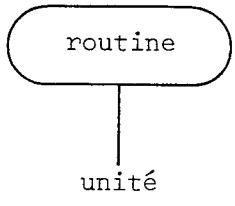


La routine $\mathcal{R}$ utilisant l'identificateur j, l'environ E_k est nécessaire à toute élaboration de la routine. En effet, que ce soit la valeur de j qui soit utilisée ou qu'elle serve seulement d'intermédiaire à l'accès d'une autre valeur, la cellule j doit être présente au moment de l'élaboration de $\mathcal{R}$. La portée de la routine $\mathcal{R}$ est donc celle de l'environ E_k.

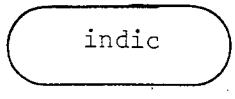
Le calcul de la portée d'une routine se fait à partir des portées des objets qu'elle utilise. Ce calcul peut être exprimé à l'aide d'attributs. Plutôt que de surcharger les mécanismes de décoration avec cette nouvelle description, nous en donnons seulement le principe.

L'identification des indicateurs, opérateurs et identificateurs leur associe un environ de déclaration qui définit leur portée. La portée est représentée par le degré d'imbrication statique de l'environ correspondant. Plus ce degré est élevé, plus la portée est restreinte.

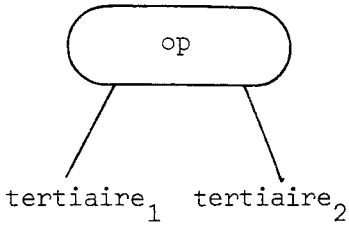
L'attribut PORTEE est synthétisé. Nous donnons maintenant l'essentiel du système d'attributs qui permet de le calculer.



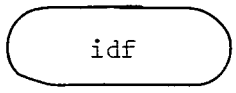
PORTEE.routine ← PORTEE.unité
portée de (routine) ← PORTEE.routine



PORTEE.indic ← portée de indicateur de (indic)
co prend en compte seulement les indicateurs à partie dynamique, puisque les autres n'apparaissent pas dans l'arbre co

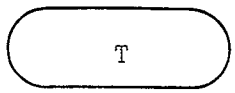


PORTEE.op ← max(portée de opérateur de (op),
PORTEE.tertiaire$_1$,
PORTEE.tertiaire$_2$)



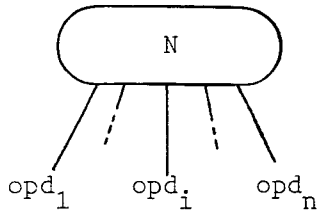
PORTEE.idf ← portée de identificateur de (idf)

pour tout autre noeud terminal



PORTEE.T ← 1

pour tout autre noeud n-aire N



PORTEE.N ← max$_i$(PORTEE.opd$_i$)

6 - DEFINITION DES FONCTIONS ET OPERATEURS UTILISES DANS LA DESCRIPTION -

```
fonction mode élément = (type mode) type
          co retourne le type des éléments du tableau mode
          co
finfonction

fonction mode paramètre = (type mode, ent i) type
          co retourne le mode du i ème - paramètre de la procédure mode
          co
finfonction

fonction mode champ = (type mode, union(ent, identificateur) i) type
          co retourne le type du champ de la structure mode repéré par
             son numéro d'ordre ou par son sélecteur i
          co
finfonction
```

```
fonction modifier = (force de contexte FC, type PRE, POST)
                    (type, modification)
    co la fonction modifier a pour paramètres la force de contexte, le mode
       a posteriori et le mode a priori du noeud.
       Son rôle est de trouver un chemin entre le mode a priori de la valeur
       et son mode a posteriori. A ce chemin, s'il existe, correspond une
       liste de modifications, qui va constituer une décoration du noeud.
       Le résultat de la fonction est constitué de deux valeurs :
       - un mode, qui est le mode a posteriori (complété s'il était incomplet)
         ou le mode erreur si aucun chemin n'existe entre PRE et POST.
       - une liste de modifications, éventuellement vide, correspondant au
         chemin entre PRE et POST.
    co
finfonction

fonction identifier = (ent no idf, bloc HBLOC) identificateur
    co cherche l'identificateur caractérisé par no idf dans l'ensemble
       de déclarations hérité HBLOC
    co
finfonction

opérateur contenu dans = (type mode1, mode2) bool
    co retourne vrai si mode1 est un des éléments de l'union mode2
    co
finopérateur

fonction complet = (type mode) bool
    co vrai si le mode n'est pas simple préfixe rep proc ou union co
finfonction

fonction mode opérande gauche = (opérateur p) type
    co retourne le mode de l'opérande gauche de l'opérateur p co
finfonction

fonction mode opérande droit = (opérateur p) type
    co retourne le mode de l'opérande droit de l'opérateur p co
finfonction

fonction mode résultat = (opérateur p) type
    co retourne le mode du résultat de l'opérateur p co
finfonction
```

```
fonction mode identificateur = (identificateur i) type
     co retourne le mode de l'identificateur i co
finfonction

fonction préfixe = (type mode) type
     co retourne le préfixe proc, rep, struct, rang, union selon la
        nature du mode
     co
finfonction

fonction mode tranche = (type pre, arbre liste indice) type
     co retourne le mode de la tranche
        Partant du mode du primaire, pre, une dimension est enlevée pour
        tout indice de la liste qui est une simple unité
     co
finfonction
```

II.3.5 - CONCLUSION -

Il faut noter que les systèmes d'attributs proposés pour les phases de décoration sont parfois incomplets. Ainsi, l'attribut synthétisé PRE est omis sur les noeuds qui ne peuvent apparaître qu'avec la force de contexte *fort* car dans ce cas il est inutile. Or, pour satisfaire aux conditions de cohérence du système d'attributs, il faudrait définir l'attribut PRE sur tous les noeuds, quitte à lui donner une valeur arbitraire. Nous avons chisi de l'omettre là où il n'était pas nécessaire à la logique de la description.

Le système d'attributs qui décrit le traitement des blocs et le traitement des modifications est fortement inspiré de la W-grammaire du Rapport. Les attributs correspondent aux métanotions, et la conversion du système d'attributs en W-grammaire est simple et systématique. L'inverse n'est pas toujours vrai et nous constatons que la transformation de la W-grammaire en systèmes d'attributs s'est avéré difficile là où le squelette hors-contexte est abandonné, ce qui conduit,à notre avis,à une perte de clarté pour le lecteur. L'intérêt d'une description par attributs est de rester proche d'une implantation,tout en convenant à la définition du langage.

II.4 - SEMANTIQUE ET GENERATION DE CODE -

Avant de donner une description détaillée de la génération de code à partir de l'arbre abstrait nous allons, dans ce chapitre, situer ce problème. Tout d'abord, nous précisons le cadre dans lequel s'exécute le code produit. Ensuite, nous discutons certains aspects de la sémantique du langage,que nous jugeons fondamentaux pour la traduction. Enfin, après avoir indiqué le schéma général de cette traduction, nous définissons la "classe" de calculateurs devant servir à l'exécution des programmes compilés.

II.4.1 - NOTIONS POUR L'EVALUATION D'UN PROGRAMME DANS UNE MACHINE PHYSIQUE -

1 - INTRODUCTION, RAPPELS -

Il existe, à l'heure actuelle, un grand nombre d'articles et d'ouvrages traitant des méthodes de définition des langages de programmation [COUSOT]. Certains d'entre eux envisagent aussi le problème de l'implantation de ces langages. Pour cela, ils utilisent, généralement, une machine idéale dont la définition cache les véritables problèmes des écrivains de compilateurs. Les calculateurs actuels étant très éloignés de ces machines définies pour les besoins des théoriciens, la réalisation pratique d'une telle approche est compromise par les éternels problèmes d'efficacité.

Dans ces conditions, on comprend aisément que chaque définition d'un nouveau langage pose des problèmes aux écrivains de compilateurs. De ce fait, il existe toujours un délai non négligeable entre la définition d'un langage et la sortie des premières implantations.

Au fur et à mesure des réalisations, certaines notions et les techniques associées sont devenues classiques. La famille des langages à structure de blocs (langages de type Algol) illustre bien cette démarche.

En effet, depuis le langage Algol 60 [RAN-RUSS] tout ce qui concerne la représentation et la gestion (à l'exécution) de la structure de blocs ne soulève plus que des difficultés très localisées.

C'est pourquoi, nous supposons que le lecteur possède les notions classiques concernant le système à l'exécution associé à ces langages [GRIES] [WICHMA-1].

Dans ce chapitre, nous allons préciser les particularités de notre implantation, dues soit au langage Algol 68, soit aux solutions parfois inhabituelles que nous avons adoptées.

2 - ASPECTS D'UN SYSTEME A L'EXECUTION POUR LE LANGAGE ALGOL 68 -

Ce chapitre, s'il se voulait exhaustif, serait très volumineux.
Il devrait traiter de nombreux problèmes (comme ceux concernant les prologues et épilogues par exemple) -intéressants techniquement-, mais n'apportant pas d'éléments fondamentaux pour la méthodologie de compilation que nous voulons expliciter. C'est pourquoi, nous avons préféré nous limiter à quelques problèmes importants. Ils concernent, plus particulièrement, les valeurs manipulées par un programme et le cadre dans lequel cette manipulation a lieu, compte tenu des caractéristiques du langage Algol 68.

2.1 - Définitions -

D'une manière générale, les langages algorithmiques de programmation sont définis pour manipuler des valeurs.

Tout compilateur est chargé de produire du code réalisant cette manipulation. Pour ce faire, il est nécessaire de représenter les valeurs et de disposer d'un cadre dans lequel ces représentations s'insèrent (le système à l'exécution, qui correspond à l'environnement dans la définition du langage). La principale difficulté est de s'assurer que le code engendré pour la manipulation des valeurs est bien conforme aux règles sémantiques du langage.

La compilation, bien que faisant appel aux mêmes notions que la définition des langages, les considère sous un aspect pratique qu'il est bon de préciser.

La manipulation de valeurs dans un programme nécessite la déclaration d'objets "possédant" ces valeurs. Pour représenter une valeur il faut disposer d'un emplacement mémoire d'une certaine taille. Ce n'est qu'une fois structuré en fonction du mode de la valeur, que cet emplacement peut être utilisé pour contenir la représentation de la valeur. De ce fait, la relation "posséder" revient à établir un lien entre l'objet déclaré dans le programme (identificateur, opérateur ...etc...) et l'emplacement destiné à contenir la représentation de la valeur. En réalité, dès que l'espace est structuré, il est considéré comme contenant la représentation d'une valeur : la valeur par défaut. Pendant l'exécution, ceci permet de vérifier, à chaque utilisation de la valeur possédée par un objet, que l'emplacement correspondant a bien été initialisé avec une "vraie" valeur.

Il existe, dans les langages de programmation, deux sortes de valeurs :

- celles dont la taille de la représentation est connue à la simple vue du mode de la valeur.

- celles dont la taille de la représentation dépend non seulement du mode mais aussi de valeurs manipulées dans le programme (ces valeurs ne sont connues qu'à l'exécution). On les appelle généralement des valeurs dynamiques.

Cette distinction, au niveau de la définition, réapparait lors de la représentation des valeurs. L'un des objectifs d'un compilateur est de gérer statiquement (c'est-à-dire à la compilation) l'espace servant à la représentation des valeurs possédées par les objets du programme. Ceci ne pose pas de difficultés pour les valeurs non dynamiques puisque leur représentation est de taille statiquement connue. Par contre, la représentation d'une valeur dynamique ne peut être traitée de la même manière. En conséquence, elle est formée de deux parties :

- la partie statique, de taille connue à la compilation, contenant toutes les informations statiques (concernant la valeur et sa représentation). Cet espace est géré statiquement (comme celui pour la représentation des valeurs non dynamiques).

- la partie dynamique, formée de la représentation de l'ensemble des valeurs (dont le nombre n'est connu qu'à l'exécution) composant la valeur dynamique. Cette partie dynamique n'est accessible que depuis la partie statique correspondante.

Lorsque -dans les chapitres suivants- nous ne nous intéressons qu'à la partie statique de la gestion de l'espace, nous employons le terme "représentation statique" pour parler aussi bien de la représentation des valeurs non dynamiques,que de la partie statique de la représentation des valeurs dynamiques.

2.2 - Elaboration collatérale -

2.2.1 - La notion -

L'élaboration de certaines constructions du langage peut nécessiter préalablement les élaborations indépendantes de plusieurs valeurs.

Ceci peut se produire aussi bien pour réaliser le traitement sémantique associé à une construction (c'est le cas, par exemple de l'affectation, avec les valeurs de la source et de la destination) que pour construire la valeur retournée par cette construction (c'est le cas, par exemple, d'un appel de procédure, avec les valeurs du primaire et de tous les paramètres).

Dès qu'il est nécessaire d'élaborer indépendamment plusieurs valeurs, le problème se pose de savoir dans quel ordre il faut le faire. Il est bien évident, en effet, que les résultats obtenus ne sont pas les mêmes, suivant l'ordre choisi, si l'élaboration de l'ensemble de ces valeurs met en jeu des effets de bord.

C'est pour ces situations qu'à été introduite en Algol 68 la notion d'ordre collatéral d'élaboration (ou élaboration collatérale, par abus de langage). Cette notion signifie qu'il n'existe pas d'ordre précis pour réaliser l'élaboration des valeurs. En conséquence, quel que soit l'ordre dans lequel cette élaboration est finalement faite, l'ensemble des valeurs obtenues doit toujours être le même. Si ce n'est pas le cas, le résultat est dit "indéterminé".
Signalons que cette notion ne doit pas être confondue avec la notion de parallélisme, telle qu'elle est définie dans le langage. Dire que plusieurs valeurs sont élaborées collatéralement signifie que leurs élaborations sont à réaliser dans un ordre quelconque indépendamment les unes des autres (ce qui les empêche, notamment, de s'interrompre mutuellement). Dire que plusieurs valeurs sont élaborées en parallèle signifie que l'utilisateur contrôle leurs élaborations, par exemple au moyen de sémaphores. Dans ce cas, leurs élaborations peuvent ne pas être indépendantes mais coordonnées par l'utilisation de ces sémaphores. Ce parallélisme pourrait être qualifié de "parallélisme algorithmique" ou de "parallélisme logiciel" (software)" par opposition au "parallélisme machine" ou "parallélisme matériel (hardware)" qui correspond mieux à la notion d'évaluation collatérale.

2.2.2 - L'élaboration collatérale et le compilateur -

Pour un compilateur, cette notion est très intéressante à deux niveaux :

1) Puisque pendant l'exécution, il faut bien faire l'élaboration de ces valeurs en position collatérale, un ordre doit, finalement, être fixé par le compilateur. Ayant toute liberté vis-à-vis de cet ordre, le compilateur peut en choisir un, suivant des critères d'implantation sans rapport sémantique

avec le langage (plus grande simplicité de la phase de génération, meilleure efficacité du code généré, meilleure utilisation des ressources).

2) En principe, chaque fois qu'une construction fournit une valeur (soit à partir d'une valeur déjà existante, soit "possédée" par un objet du programme), il faudrait créer un nouvel exemplaire de cette valeur, ce qui réalise une protection empêchant toute modification ou destruction de cette valeur.

Dans l'hypothèse où cette protection n'est pas réalisée, examinons ce qui se passe dans les deux cas possibles d'utilisation de la valeur :

- la valeur n'est pas utilisée en position collatérale. Dans ce cas, il ne peut y avoir altération de cette valeur puisqu'aucune autre élaboration n'est faite avant son utilisation.

- la valeur est utilisée en position collatérale. Si l'évaluation d'une des autres valeurs (dans la même position collatérale) la modifie ou la détruit, nous avons un effet de bord et (d'après ce que nous avons dit précédemment) le résultat est indéterminé.

Remarque :
Il existe des cas où les effets de bords sont indécelables statiquement (à la compilation) ce qui est regrettable pour l'utilisateur.

Exemple : Soit le programme :

```
début
   ent a := 3 ; réel b ;
   b := a/2
fin
```

La valeur *3* apparaissant dans *ent a := 3* n'est pas en position collatérale. Quelle que soit la construction se trouvant à la place de la constante *3*, la valeur entière finalement délivrée par cette construction ne peut être altérée avant d'être utilisée pour "initialiser" la variable *a*.

Il n'en est pas de même pour la formule *a/2* dans laquelle les deux opérandes sont en position collatérale. Si à sa place nous trouvions *(a:=a+1 ; a) /a* la valeur finale de cette opération dépendrait de l'ordre d'élaboration des deux opérandes.

En conséquence, le fait de manipuler collatéralement plusieurs valeurs permet de ne pas protéger systématiquement ces valeurs. Ceci améliore les performances du code engendré pour un programme.

Remarque :
Parmi les constructions nécessitant l'élaboration collatérale de plusieurs valeurs, l'instruction d'affectation joue un rôle particulier. Une fois le traitement réalisé, la valeur retournée est celle de la destination. Considérons alors l'exemple :

```
début
   [1:4] ent t := (1,2,3,4) ;
   t [2:4] := t[1:3]
fin
```

Si nous ne prenons aucune précaution en produisant le code pour l'affectation *t[2:4] := t[1:3]* les éléments du tableau risquent d'avoir tous la valeur 1 à la fin de l'exécution du programme, bien qu'il n'y ait pas d'effet de bord. Un tel résultat n'est bien sûr pas conforme à la sémantique de l'affectation. Ceci montre les lacunes des calculateurs actuels dès qu'il s'agit de manipuler des valeurs structurées. Une telle situation ne peut pas se produire avec les autres constructions car elles créent explicitement une valeur à partir des valeurs élaborées collatéralement.

Pour la description du processus de génération de code, nous supposons qu'il est possible de produire des instructions manipulant globalement toute valeur Algol 68 quelles que soient ses caractéristiques. Ceci préserve la généralité du principe que nous avons énoncé sur l'inutilité de protéger les valeurs en position collatérale.

2.3 - Système à l'exécution -

Il n'est pas question dans ce qui suit de parler longuement du système à l'exécution mais plutôt de rappeler certains aspects importants et de souligner les traits caractéristiques du système que nous avons défini.

2.3.1 - Organisation générale de la mémoire. Pile et tas -

La durée de vie de chaque valeur Algol 68 existant dans un programme peut être :

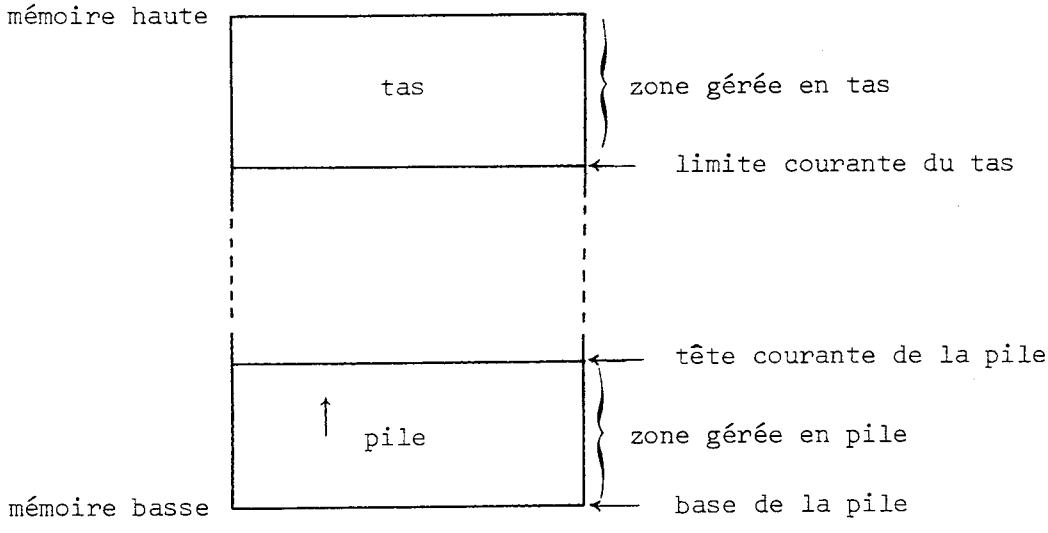
- soit liée à la structure de blocs du programme,
- soit globale, c'est-à-dire égale à celle du programme principal. Cette valeur cesse d'exister lorsqu'il n'est plus possible d'y accéder.

Nous parlons dans la suite de valeurs locales et de valeurs globales.

Une organisation de la mémoire sous forme de pile n'est pas suffisante pour prendre en compte ces deux aspects. Ayant décidé d'utiliser une gestion classique pour les valeurs locales, nous considérons la mémoire comme formée de deux parties :

- la pile servant à la représentation des valeurs locales,
- le tas servant à la représentation des valeurs globales.

Ces deux zones coexistent en mémoire de la manière suivante :



Les mécanismes d'allocation et de récupération d'espace sur une pile continue sont devenus classiques [RAN-RUSS] et nous n'évoquons, dans le paragraphe suivant que quelques points particuliers.

L'allocation d'espace sur le tas se fait par décrémentation, de la limite courante du tas, d'une valeur égale à la taille demandée.

La durée de vie des valeurs s'y trouvant étant globale, la place occupée par la représentation d'une valeur ne peut être récupérée que lorsque cette représentation n'est plus accessible par le programme. Ne disposant pas aisément de cette information pour chaque valeur représentée dans le tas, il n'existe pas de mécanisme simple récupérant l'espace, dès qu'une valeur est devenue inutile.

La pile et le tas se partageant l'espace mémoire, il peut arriver que la limite courante du tas et la tête courante de pile se rencontrent. Il peut, à cet instant, exister dans le tas des emplacements devenus inaccessibles. La récupération de ces "trous" et le retassement (vers le haut) des emplacements utilisés du tas (accompagné de la mise à jour des accès à ces emplacements) va faire remonter la limite courante du tas et fournir une zone de mémoire libre entre la pile et le tas. Le conflit d'espace entre la pile et le tas étant résolu, l'exécution du programme peut être poursuivie. Bien entendu, lorsqu'il n'est plus possible de récupérer un minimum d'espace, l'exécution est interrompue.

2.3.2 - Organisation et gestion de la pile - Environnements -

Pour un langage à structure de blocs, la gestion de la pile est, en principe, faite par région (une région correspond à un bloc ou à un corps de procédure). Dans ces conditions un environnement est créé dans la pile à chaque entrée de région, et libéré (ce qui récupère l'espace alloué dans la pile, pour la représentation des valeurs de la région) à chaque sortie de région.

Nous considérons un environnement comme étant constitué de deux parties :

- la zone statique de taille connue à la compilation,

- la zone dynamique servant à la représentation des valeurs allouées dynamiquement. Sa taille peut varier à tout instant de l'exécution.

Dans la zone statique, nous distinguons quatre groupes :

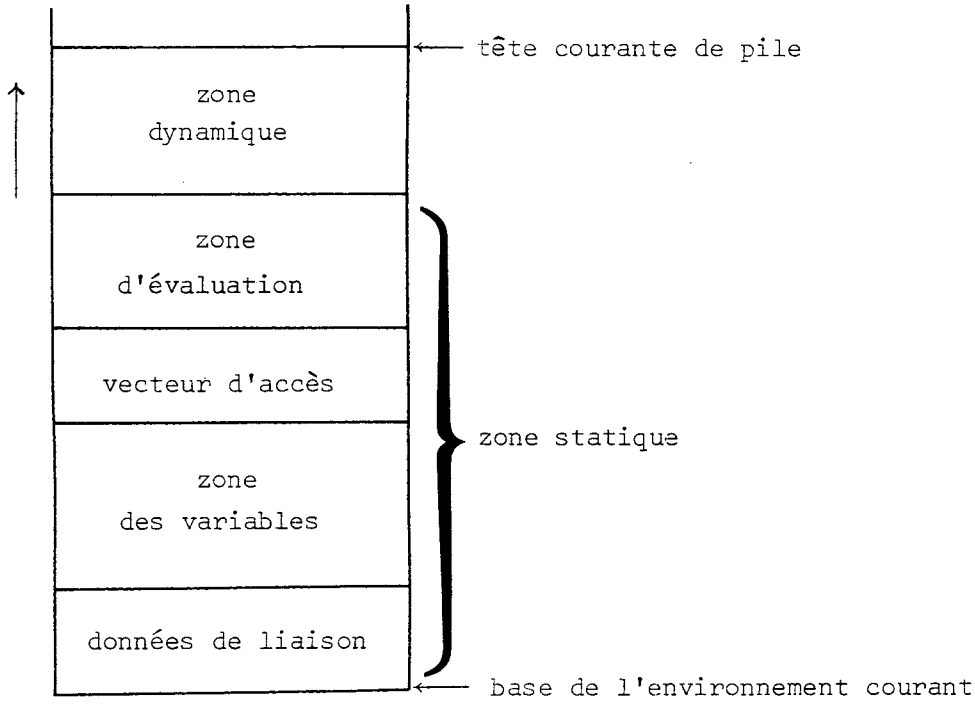
- les données de liaison, de taille fixe, contenant les informations nécessaires à la gestion de la pile (chaînage dynamique, adresse de retour, pointeur tête de pile ...etc...),

- la zone des variables contenant la représentation des parties statiques des valeurs possédées par les objets déclarés dans la région. La gestion des emplacements de cette zone est faite à la compilation.

- le "vecteur d'accès" aux représentations des valeurs. Très souvent appelé "display" ce vecteur contient les bases des environnements accessibles depuis l'environnement courant. Son nombre d'éléments ne dépend que de l'imbrication statique de la région,

- la zone d'évaluation. Elle sert aux sauvegardes temporaires des représentations statiques des valeurs. Elle est gérée statiquement pendant la génération de code.

Dans ces conditions, un environnement a la structure suivante :



La création d'un nouvel environnement en début d'un bloc est plus simple qu'en début de routine. En effet, les données de liaison sont simplifiées (il n'y a pas d'adresse de retour, notamment) et le vecteur d'accès est formé de celui de l'environnement précédent augmenté de l'accès à l'environnement courant (ce qui n'est pas aussi simple pour les routines).

C'est pourquoi, nous avons préféré (comme la majorité des écrivains de compilateurs) ne faire la gestion de la pile qu'au niveau des routines. En conséquence la zone statique allouée en début de chaque routine tient compte de toute la place statique nécessaire pour les blocs internes à cette routine. La répartition, entre les différents blocs internes, de l'espace prévu pour la zone des variables, ne dépend que de l'imbrication de ces blocs ; ceci permet d'effectuer cette répartition à la compilation.

Remarque :

Certains auteurs préfèrent considérer les blocs comme étant des routines anonymes sans paramètre, ou ayant comme paramètres, les variables externes utilisées.

Exemple: dans le contexte des objets particuliers manipulés par le système MULTICS [CLARK], le programme PL/1 suivant :

```
DCL  J...
BEGIN
  DCL  I...
  ...
  I * J
  ...
END ;
 :
```

peut être considéré comme étant:

```
DCL  J...
ALPHA : PRØC (J)
  DCL  I...
  ...
  I * J
  ...
END ALPHA ;
CALL ALPHA (J) ;
```

L'espace sur la zone d'évaluation étant récupéré statiquement à chaque point-virgule, la gestion de cette zone ne dépend pas de la structure de blocs et n'est, donc, pas affectée par ce changement.

La zone dynamique va présenter quelques différences. En effet, la récupération d'espace ne se faisant qu'en sortie de routine, il peut exister sur la zone dynamique des représentations de valeurs devenues inaccessibles par suite de la fermeture de blocs internes à la routine. Cet espace est, de toute façon, récupéré lorsque l'exécution du texte de la routine est terminé [WICHMA-1].

Remarque :
Le choix d'une telle gestion peut être une gêne technique pour des (mauvais) programmes ayant, par exemple, la structure suivante :

```
début co bloc A co
   co petit nombre de déclarations demandant peu d'espace dans la zone des
      variables co
   ...
fin ;
...
début co bloc B co
   co très grand nombre de déclarations co
   ...
fin ;
 :
```

Pour les blocs disjoints A et B appartenant à une même routine, les zones des variables se recouvrent. Dans ces conditions l'allocation d'espace en début de routine va être fonction de la plus grande de ces zones (celle du bloc B). Il se peut que pour certain jeu de données il n'y ait jamais exécution du bloc B et que l'allocation d'espace en début de routine ne soit pas possible à cause de la trop grande taille de la zone des variables.

2.4 - Valeurs -

2.4.1 - Représentation des valeurs -

Nous ne donnons ici que les aspects fondamentaux des représentations des valeurs dont le mode est obtenu en utilisant les règles de composition données dans le langage. La représentation des modes de base (ex : *ent*, *réel*, *bool*, ... etc...) est étroitement liée à la machine et ne présente pas d'intérêt particulier au point de vue fonctionnel.

1) Valeurs de mode repère

Elles sont représentées par un pointeur vers la représentation de la valeur constituante. Pour des raisons d'efficacité les valeurs de mode repère introduites par les déclarations de variable sont représentées sous une forme abrégée. Nous examinons dans la troisième partie ce problème plus en détail.

2) Valeurs de mode procédure

Ces valeurs étant manipulables au même titre que les autres valeurs Algol 68, elles sont représentées par deux pointeurs :

- l'un est l'adresse du code engendré pour la routine associée
- l'autre contient la base de l'environnement nécessaire pour exécuter la routine associée.

3) Valeurs de mode tableau

Il s'agit de valeurs dynamiques qui sont donc représentées en deux parties.

La partie statique de la représentation est formée, entre autres ,

- d'un pointeur appelé origine virtuelle permettant d'accéder à la partie dynamique
- d'autant de triplets que le tableau a de dimensions.

Chaque triplet comprend la borne inférieure, la borne supérieure et l'enjambée de la dimension correspondante.

La partie dynamique est constituée de l'ensemble des représentations des éléments du tableau.

4) Valeurs de mode structure

Ces valeurs peuvent avoir des parties dynamiques.
La représentation statique est formée de la juxtaposition des représentations des parties statiques des champs.

La représentation des parties dynamiques (si elles existent) est formée de l'ensemble des représentations des parties dynamiques des **valeurs** constituantes.

5) Valeurs de mode union

Ces valeurs peuvent avoir des parties dynamiques.
A tout instant une telle valeur est de mode égal à un de ses modes composants.

La représentation statique est formée :

- d'un en-tête contenant la caractérisation du mode courant de la valeur
- de la représentation statique de la valeur courante (en utilisant le principe de recouvrement).

La représentation des parties dynamiques (si elles existent) est formée de celle des parties dynamiques de la valeur courante.

2.4.2 - Allocation des représentations -

Suivant le type d'allocation demandée, les représentations de valeurs vont se trouver, soit dans la pile, soit dans le tas.

Dans le cas d'une représentation à créer dans le tas, l'établissement de la relation "posséder" oblige à avoir, dans la zone statique de l'environnement courant, la racine de la représentation. Dans ces conditions, toutes les représentations de valeurs dans le tas sont accessibles initialement depuis les zones statiques.

D'autres valeurs, dont le changement de taille de la représentation n'est pas compatible avec la gestion de la pile, sont représentées comme les valeurs globales. C'est le cas des tableaux flexibles et des unions lorsque la valeur courante est une valeur dynamique.

Exemple :

```
début
   union (ent, [ ] réel) uetr ;
   [1:10] réel t ;
   [5:20] réel tp ;
     ⋮
   uetr := t ;
     ⋮
   uetr := tp ;
     ⋮
fin
```

Lors de l'élaboration de la déclaration de *uetr*, il n'est pas possible de réserver l'espace nécessaire pour les éléments d'un tableau de valeurs réelles puisque les bornes ne sont pas spécifiées dans le déclareur. Lors de l'exécution de l'affectation *uetr := t*, de l'espace est alloué pour les éléments du tableau (10 valeurs réelles). De la même manière, de l'espace est alloué lors de l'exécution de *uetr := tp* (15 valeurs réelles).
Les représentations des éléments d'un tableau doivent rester accessibles aussi longtemps que la représentation statique du tableau. Il n'est donc pas possible de faire les allocations dans l'environnement courant, car il peut être interne à celui contenant la représentation statique correspondante.

En faisant les allocations dans le tas on est sûr que la durée de vie des éléments est suffisante.
Le problème est identique avec les tableaux flexibles.

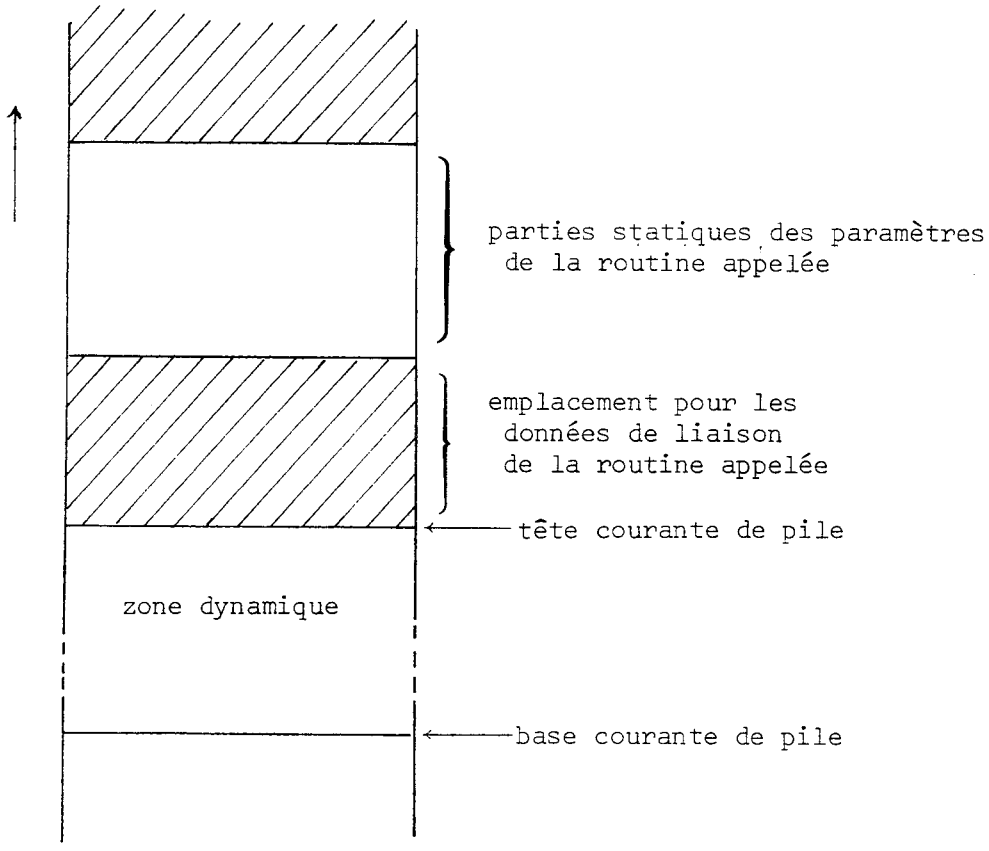
2.4.3 - Transmission de valeurs entre environnements -

Cette transmission se limite aux paramètres et aux résultats de routines.

1) Paramètres

Les représentations statiques des paramètres (s'ils existent) d'une routine vont se trouver dans les premiers emplacements de la zone des variables de la routine. En examinant la structure d'un environnement, on s'aperçoit qu'elles sont ainsi rangées tout de suite après les données de liaison.
La taille de ces données étant la même pour toutes les routines, les paramètres sont toujours au même déplacement à l'intérieur de chaque environnement.
A l'exécution, nous connaissons à tout instant la valeur du pointeur sur la tête de pile. De ce fait, depuis chaque point d'appel, nous rangeons les représentations statiques des paramètres effectifs (arguments) à l'emplacement associé aux paramètres formels (paramètres) dans l'environnement de la routine à appeler (la base de l'environnement étant la valeur du pointeur sur la tête de pile au moment de l'appel). Une fois les données de liaison mises en place et l'allocation pour la zone statique faite, la première action réalisée à l'intérieur de la routine est la transmission (avec allocation d'espace dans la zone dynamique) des parties dynamiques (si elles existent) des paramètres.

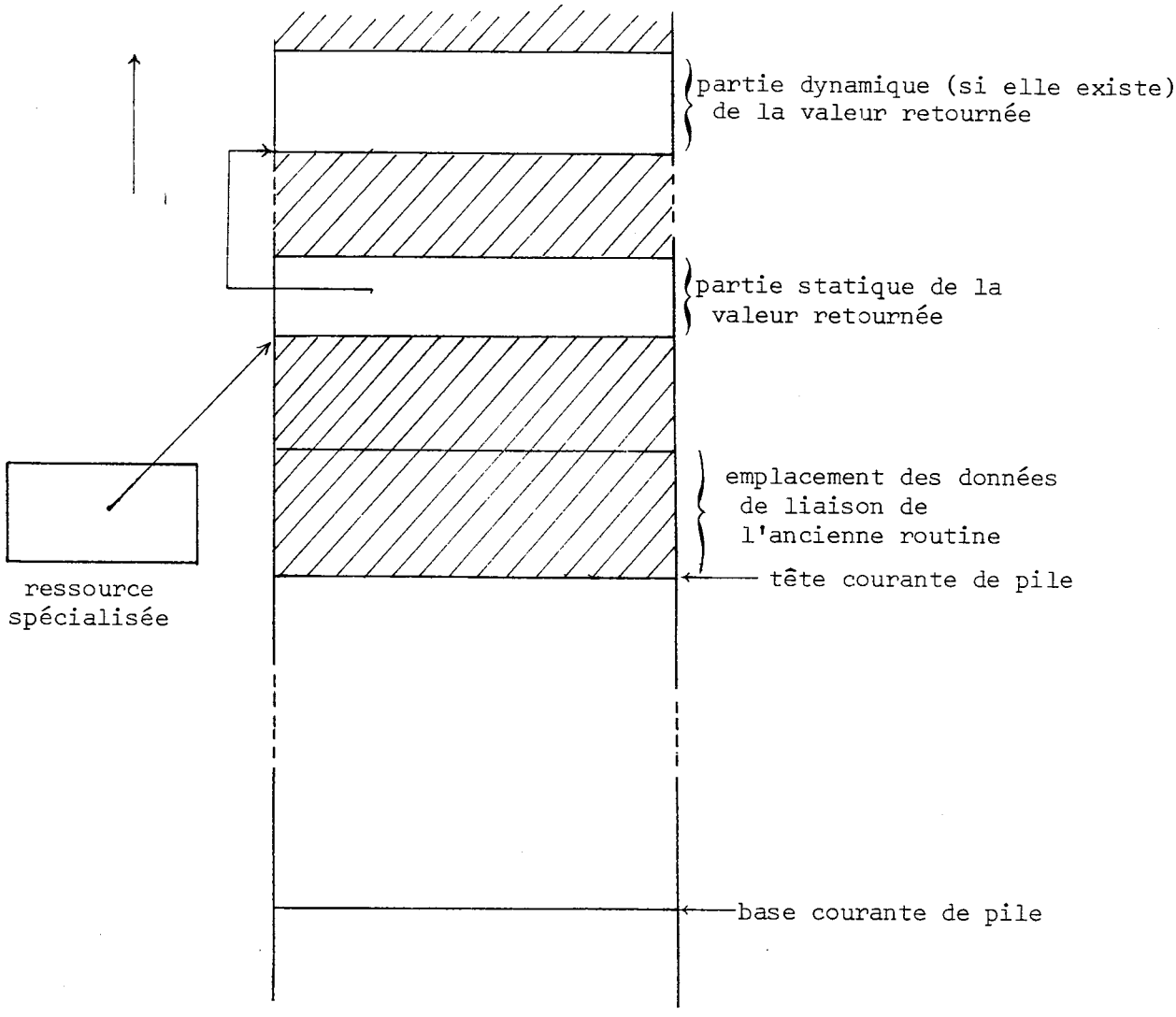
En définitive, la transmission d'un paramètre crée un nouvel exemplaire de la valeur. Ceci revient à réaliser la protection de toute valeur servant de paramètre effectif (d'argument).
Juste avant l'exécution du branchement au code de la routine, nous avons la configuration suivante :



2) Résultats

Pour que la valeur retournée par une procédure soit utilisable une fois l'exécution du programme revenue au point d'appel, il est nécessaire de transmettre cette valeur à l'environnement d'appel. C'est pour cela que l'accès à sa représentation est transmis au moyen d'une ressource spécialisée. Dès le retour au point d'appel, il y a recopie dans la zone d'évaluation de la représentation de la partie statique de la valeur. Ceci permet de libérer la ressource spécialisée et d'avoir, comme pour les autres valeurs, la représentation de la partie statique accessible statiquement dans la pile.

Souvent, la représentation de la valeur à retourner comme résultat est dans un environnement accessible depuis le point d'appel, mais il peut arriver que cette valeur ait une représentation locale à l'environnement de la routine qui la délivre (elle peut avoir ou non des parties dynamiques). Dans ce cas, de retour au point d'appel, nous avons la configuration suivante :



Examinons ce qui peut se passer, une fois la représentation statique recopiée dans la zone d'évaluation.

Deux cas sont possibles :

- la valeur retournée n'est pas utilisée en position collatérale.
La représentation de sa partie dynamique, bien que n'appartenant pas à la partie active de la pile, reste accessible et ne peut pas être écrasée avant son utilisation.

- la valeur retournée est utilisée en position collatérale. Dans ce cas, avant qu'elle soit effectivement utilisée, il va y avoir d'autres élaborations et, peut-être, des demandes d'espace sur la pile, ce qui risque d'écraser la représentation de la partie dynamique. Il faut donc la transférer dans l'environnement d'appel.

C'est pourquoi, de retour au point d'appel, il y a recopie non seulement de la partie statique, mais aussi de la partie dynamique de la représentation de la valeur retournée pour préserver son accessibilité. Cette recopie peut être onéreuse et nous verrons dans la troisième partie qu'elle peut être évitée dans la majorité des cas, en ne la réalisant que dans les positions collatérales où risquent d'être manipulées plusieurs représentations de valeurs dynamiques.

3 - CONCLUSION -

Le système à l'exécution que nous venons de présenter reste résolument classique malgré quelques solutions inhabituelles. Nous n'avons pas voulu entrer dans le jeu d'une définition (plus ou moins formalisée) faisant abstraction du système à l'exécution, afin de montrer que de tels systèmes sont, malgré quelques faiblesses, assez bien adaptés à l'exécution des programmes écrits en Algol 68.

II.4.2 - SEMANTIQUE ET ARBRE ABSTRAIT -

1 - INTRODUCTION -

1.1 - Squelette sémantique -

La génération de code a pour but de produire du code dont l'exécution donne le même résultat que celui fourni par l'interprète utilisé pour la description de la sémantique dans le Rapport de définition du langage [VANWIJ-2]. Les informations sémantiques, nécessaires à l'élaboration du programme par cet interprète, doivent donc être décrites pendant la phase de génération. Nous travaillons sur l'arbre abstrait décoré défini dans les chapitres précédents. Cet arbre reflète la structure profonde du programme et constitue le "squelette sémantique" nécessaire à la traduction du programme [SINTZOFF]. Ce squelette précise ponctuellement les "calculs" à effectuer et va servir à les enchaîner.

Remarquons qu'il est alors aisé de prendre en considération la notion de calcul collatéral, car les "squelettes" correspondant aux constructions en position collatérale sont enracinés au même niveau dans le squelette principal. L'enchaînement des calculs devant être celui spécifié dans le Rapport, les règles de construction du squelette sont déduites des paragraphes du Rapport traitant de la sémantique.

1.2 - Principe -

Nous avons voulu mettre en oeuvre un processus préservant la localité de la traduction. Pour cela nous avons adopté le principe suivant :

Pendant le parcours (de l'arbre abstrait) dirigeant la phase de génération de code, les informations décrivant les valeurs qui vont être manipulées à l'exécution par le code associé à un noeud de l'arbre, sont transmises depuis les fils de ce noeud. Ces informations sont construites et transmises en même temps qu'est produit le code correspondant à l'élaboration des fils.

Exemple :

Avant d'engendrer le code réalisant une affectation, nous produisons le code élaborant les valeurs de la source et de la destination. Un "descripteur" de compilation décrivant chacune de ces valeurs est synthétisé depuis les sous-arbres correspondants jusqu'au noeud (affectation) . Sur ce noeud nous disposons alors de toutes les informations nécessaires à la génération du code de l'affectation.

Remarquons qu'une fois engendré le code pour un noeud, il n'est plus nécessaire de conserver ce noeud car :

- d'une part le traitement associé à ce noeud est terminé,

- d'autre part le traitement associé à ses sous-arbres est, d'après le principe précédent, également terminé.
Seul un descripteur décrivant la valeur (si elle existe) retournée par la construction est synthétisé vers le noeud père.

Sans vouloir entrer dans les détails, on peut remarquer l'analogie entre ce parcours de "réduction" d'un arbre abstrait et les processus d'interprétation d'un programme abstrait.

2 - SEMANTIQUE DES VALEURS -

2.1 - Définition -

Cet aspect de la sémantique n'apparait pas clairement dans la définition formelle de la sémantique des langages classiques. Dans la définition du langage Algol 68, les structures de données sont l'objet d'une attention particulière (par rapport aux autres langages) ce qui met en évidence cet aspect de la sémantique que nous appelons "sémantique des valeurs".

Elle regroupe deux notions liées aux valeurs :

a) - la structure d'une valeur (structure de la donnée) et le parcours à l'exécution de cette structure.
En pratique, la représentation d'une valeur devant refléter la structure de cette valeur, nous parlerons du parcours de la représentation d'une valeur au lieu du parcours de la structure d'une valeur. Ce parcours est invariant quel que soit le point du programme à partir duquel on l'entreprend. il ne dépend que du mode de la valeur.

Exemple :
Considérons le programme :

```
début
   ent i,j,k ;
    :
   [i:j] struct (ent a, [i:k] réel v) t,u ;
    :
   t := u ;
    :
fin
```

L'opération d'affectation $t := u$ nécessite la manipulation des valeurs formant la source et la destination. Ces valeurs sont structurées.

L'affectation d'un tableau (d'une structure) consiste à affecter chaque élément (champ) du tableau (de la structure).

En conséquence *t := u* revient à affecter :

j-i+1 éléments de mode *struct (ent a, [i:k] réel v)*

c'est-à-dire :

j-i+1 valeurs entières (sélectionnées par *a*)

et :

j-i+1 champs de mode *[i:k] réel v*

c'est-à-dire :

*(j-i+1) * (k-i+1)* valeurs réelles.

Le code produit doit donc parcourir, de manière concordante, les représentations des valeurs formant la source et la destination pour accéder aux valeurs simples se trouvant dans la source et aux emplacements où doivent être rangées ces valeurs dans la destination.

b) - l'accès à la représentation de cette valeur.
Cet accès est soit statique pour les valeurs Algol 68 ne comportant pas de tableaux, soit dynamique si, dans le cas contraire, on veut accéder aux éléments du tableau. L'accès aux éléments se fait par l'intermédiaire d'un descripteur (ou partie statique) pour lequel on garde toujours un accès statique.

L'utilisateur se préoccupe en général peu de cette sémantique. Pourtant, en l'utilisant judicieusement, il peut effectuer des optimisations importantes sur un programme, surtout en Algol 68 où par l'intermédiaire des modes il a la possibilité de contrôler de manière précise la gestion des valeurs (structures de données) qu'il utilise.

De nombreuses constructions en Algol 68 entraînent des copies de valeur avec création, éventuellement dynamique, d'une allocation pour cette nouvelle valeur ; d'autres provoquent des protections de valeur avec, comme précédemment, copie et création. Les transmissions de valeurs entre environnements posent les mêmes problèmes.

Plus généralement, un des objectifs des langages de programmation est de permettre la manipulation de données (ou valeurs). Un grand nombre de constructions dans un langage n'existe que pour effectuer des opérations sur les valeurs.En Algol 68 cette caractéristique a été systématisée.

De ce fait, la plupart des constructions de base possèdent une valeur. Chacune de ces valeurs est utilisée ou abandonnée au gré du programmeur.

Exemple :

1) La valeur d'une affectation est une valeur de mode repère qui représente l'accès à l'opérande gauche (destination).
Le programmeur peut, par exemple, utiliser cette valeur, comme argument dans l'appel d'une routine, ou partie droite d'affectation, ou simplement l'abandonner.

2) Considérons le programme :

```
début
   [1:100, 1:10] ent t ;
    ⋮
   proc p = ([] ent u) neutre :
   début
        ⋮
   fin ;
   proc q = (rep [] ent v) neutre :
   début
        ⋮
   fin ;
    ⋮
   p(t) ;
   q(t) ;
    ⋮
fin
```

Lors d'un appel de procédure il y a copie des valeurs des arguments, dans l'environnement de la routine appelée.

L'appel *p(t)* va entraîner la copie de la valeur tableau passée en paramètre, c'est-à-dire de la partie statique (descripteur) et de tous les éléments. Cette copie peut éventuellement avoir lieu élément par élément (avec, dans ce cas, un parcours complet de la représentation), si une opération de tranche a été réalisée sur le tableau. On obtient ainsi un nouvel exemplaire de la même constante de mode [] *ent*, ce qui permet de protéger la valeur

initiale (le coût de l'opération étant la copie des 100 x 10 entiers).

Pour l'appel *q(t)* il n'y a copie que de la représentation de la valeur de mode *rep* [] *ent* c'est-à-dire du descripteur du tableau (l'équivalent d'une dizaine de valeurs entières).

La copie ne nécessite aucun parcours. On peut accéder de la même manière aux entiers. Le seul ennui est que dans ce cas, de l'intérieur de la routine, on peut modifier les entiers; mais existe-t-il beaucoup de cas où le programmeur sans intention délibérée modifiera les éléments du tableau ? Il y a cependant fort à penser que le déclareur d'une telle routine sera écrit dans la pratique *proc ([] ent)* et non *proc (rep* [] *ent)* même si le programmeur a la certitude de n'utiliser qu'une constante dans sa routine. C'est peut-être payer bien cher une vérification statique !

Le problème reste le même avec des structures de données plus élaborées.

En définitive, une grande importance doit être accordée à cet aspect de la sémantique,dont une partie seulement peut être traitée statiquement. Au niveau de la définition de la fonction de traduction, on s'efforce,tout en ne modifiant pas la signification des programmes, de limiter ou d'optimiser les copies de valeurs surtout pour les valeurs contenant des tableaux.

2.2 - Les valeurs et l'élaboration d'un programme -

Pendant l'élaboration d'un programme, pour pouvoir être utilisée, une valeur est soit possédée par un objet, auquel cas sa représentation est en mémoire, soit une notation et alors sa représentation est dans une table des constantes construite pendant la génération de code.

2.2.1 - Accès -

Pendant l'exécution, l'accès à la représentation d'une valeur se fait toujours par l'intermédiaire d'une partie statique, même si la valeur est dynamique. Cette partie statique se trouve soit dans la zone statique d'un environnement (rappelons que les objets globaux ont un accès à partir de la zone statique), soit dans la table des constantes. Par suite, par abus de langage, nous désignons cet accès à la partie statique de la représentation de la valeur, par accès à la valeur. Remarquons que cet accès peut toujours être géré statiquement, soit pendant les processus de décoration pour les objets ayant une identification externe (zone des variables), soit pendant la génération de code (zone d'évaluation).

Au moment de la production de code, il est commode de représenter cet accès sous forme d'une base d'environnement et d'un déplacement par rapport à cette base.

2.2.2 - Représentation et parcours de la représentation -

Les informations statiques associées à la valeur (taille de la partie statique ...etc...) sont rattachées au mode de la valeur. Si la valeur est "arborescente", (structurée), le parcours de sa représentation à l'exécution ne dépend que de son mode. Les informations statiques nécessaires à la génération du code de parcours sont données par la structure du mode de la valeur ; les informations connues dynamiquement sont rangées dans la partie statique, ce qui permet d'avoir un accès statique à ces informations.

Exemple : Soit le programme :

```
début
   ent i,j ;
    :
   [i:j] réel u,v ;
    :
   u := v ;
    :
fin
```

Pour engendrer le code parcourant la représentation de la valeur formant la source de l'affectation $u := v$ nous avons besoin, en particulier, des informations statiques suivantes :

- nombre de dimensions du tableau (1)
- mode des éléments (*réel*)
- accès à la partie statique de la représentation (déplacement d_v par rapport à la base de l'environnement courant).

A l'exécution les informations connues dynamiquement sont rangées dans la partie statique. Signalons par exemple :

- les valeurs des bornes (i et j)
- l'accès à la partie dynamique (c'est-à-dire aux éléments).

Dans ces conditions, on peut produire du code utilisant ces informations.

2.2.3 - Descripteur de valeurs -

Nous avons vu dans le paragraphe traitant du principe adopté pour la traduction (voir §1.2 de ce chapitre) qu'il était nécessaire de décrire à la compilation les valeurs que le code manipule pendant l'exécution. L'ensemble des informations caractérisant une valeur et sa représentation, gérées statiquement, sont regroupées dans un descripteur qualifié (abusivement) de descripteur de valeur.

Dans le paragraphe montrant sur un exemple simple le processus complet de traduction (voir ch. II.1), nous avons défini les descripteurs comme étant des doublets décrivant les caractéristiques de mode d'une valeur et l'accès à sa représentation.

Rappelons que la description de l'accès est formée :

- d'un déplacement par rapport à une base d'environnement actif
- d'un pointeur sur le descripteur de cette base.

Le descripteur associé à x est noté :

$$\delta x = (C_x, (d_x, \quad \delta base))$$

Au cours de la "réduction" de l'arbre abstrait, ces descripteurs sont synthétisés et servent à transmettre localement, les caractéristiques statiques des valeurs. Ces descripteurs servent comme opérandes des quadruplets produits (voir l'exemple ch.II.1). A partir de l'accès (entre autres), le traducteur peut déterminer les fonctions d'adressage (des opérandes des instructions) du code objet manipulant la valeur.

La génération du code, réalisant l'accès à la représentation d'une valeur, se trouve ainsi reportée au niveau de l'utilisation de cette valeur.

2.2.4 - Construction de valeurs et synthèse des descripteurs de valeurs -

Comme nous l'avons rappelé, la plupart des constructions Algol 68 délivrent des valeurs. Les descripteurs représentant ces valeurs sont synthétisés au cours de la réduction de l'arbre au niveau de chaque noeud. Nous allons examiner, suivant les types des noeuds de l'arbre abstrait, comment s'effectue la construction du descripteur synthétisé.

Deux cas peuvent se produire pour constituer une valeur :

- la valeur que délivre une construction est celle possédée par un objet déclaré dans le programme.

Exemple : Soit le programme :

```
début
    réel x ;
    :
    x := ..
    :
fin
```

La valeur délivrée par l'affectation est celle possédée par x.
Un descripteur, représentant cette valeur et accédant à la représentation correspondante dans la zone statique, est créé à partir des décorations de l'arbre abstrait.

- la valeur délivrée est construite à partir d'une autre valeur (valeur initiale), existant dans le contexte d'évaluation (ce contexte est représenté par un ensemble de descripteurs de valeurs accessibles au niveau du noeud).

Cette construction peut se faire de deux manières :

a) La représentation (à l'exécution) de la valeur existe déjà en totalité. Le descripteur synthétisé décrivant la valeur retournée est construit à partir du descripteur de la valeur initiale.
Dans ce descripteur, la représentation de l'accès est obtenue :

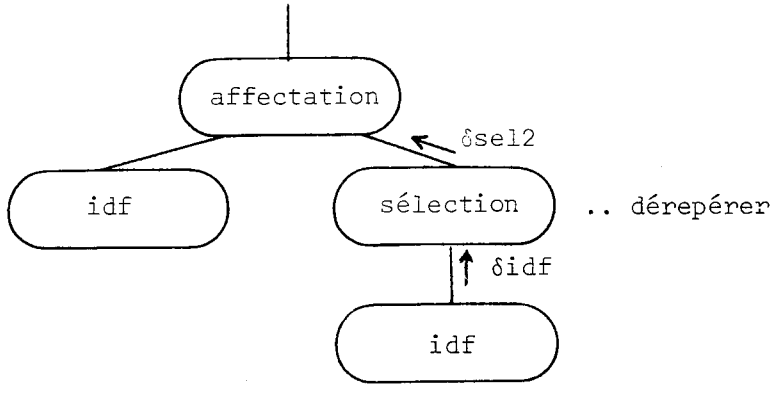
- soit en modifiant celle de l'accès à la valeur initiale.
Il n'y a alors pas de génération de code. C'est le cas pour la sélection simple d'un champ d'une valeur structurée et pour le dérepérage d'une valeur (cf. exemple ci-dessous).

- soit en décrivant un accès par l'intermédiaire d'une ressource (registre) contenant l'adresse de la valeur. Dans ce cas, on produit le code calculant et rangeant l'adresse de cette valeur dans la ressource.
On obtient ainsi un accès statique à la valeur. Ceci se produit lors d'une opération d'indexation sur une valeur de mode tableau.

Exemple : Soit le programme Algol 68 suivant :

```
début
   rep struct (ent a,b) rst = ... ;
   ent x ;
   x := a de rst ;
    :
fin
```

Considérons le sous-arbre associé à l'affectation.



Examinons le traitement du fils droit.
Sur le noeud (idf) est construit le descripteur δidf décrivant la valeur possédée par l'identificateur *rst*.

δidf = (C_{rst}, (d_{rst}, ↓))
└──→ δbase de l'environnement courant.

C_{rst} représente les caractéristiques d'une valeur de mode :
rep struct (ent a,b)

d_{rst} est le déplacement de l'espace réservé dans la zone des variables de l'environnement courant pour la valeur possédée par rst.

δidf est synthétisé jusqu'au noeud (sélection)
La valeur délivrée par cette construction est décrite par un descripteur δsel1 obtenu en :

- remplaçant C_{rst} par les caractéristiques (C_{re}) d'une valeur de mode *rep ent*.

- modifiant le déplacement d_{rst} du déplacement du champ sélectionné par *a* (d_a) dans la représentation de la valeur possédée par *rst*.

δsel1 = (C_{re}, (d_{rst} + d_a, ↓))
└──→ δbase de l'environnement courant

Pour tenir compte de la modification dérepérer il suffit de considérer la valeur décrite par δsel1 comme étant l'accès à la valeur demandée.

En conséquence, nous contruisons δsel2 décrivant la valeur entière :

δsel2 = (C_{ent}, (0, ↓))
└──→ δsel1 = (C_{re}, (d_{rst} + d_a, ↓))
└──→ δbase de l'environnement courant

L'ensemble de ces traitements n'a nécessité aucune production immédiate de code.

b) La représentation (à l'exécution) de la valeur n'existe que partiellement ou pas du tout. La valeur est constituée par transformation d'une valeur initiale appartenant au contexte d'évaluation.

Dans ce cas, on construit la partie statique de la représentation de cette valeur dans la zone d'évaluation de l'environnement courant. Son initialisation à partir de la partie statique de la valeur initiale nécessite généralement d'engendrer du code.

1) La modification élargir permet, entre autres, d'obtenir la valeur réelle correspondant à une valeur entière. Ceci nécessite de produire du code. En pratique, la représentation de la valeur finale n'est pas dans la zone d'évaluation proprement dite mais dans une ressource privilégiée du calculateur (registre). Rappelons que nous considérons la zone d'évaluation comme la "zone de débordement" de ces ressources (voir utilisation des ressources d'un calculateur : ch. II.4.4, §2).

2) Dans le cas d'une opération de "tranche" sur une valeur de mode tableau, la partie statique de la représentation de la valeur obtenue (descripteur) doit être construite à partir de celle de la valeur initiale. L'espace nécessaire est réservé dans la zone d'évaluation.

Si la valeur obtenue est dynamique il faut associer la partie statique de la représentation à sa partie dynamique.

Généralement, cette partie dynamique existe et elle est contenue dans celle de la représentation de la valeur initiale.

Exemple :

Si nous réalisons une opération de tranche sur une valeur de mode tableau, les éléments du tableau résultant sont aussi des éléments du tableau initial.

Il n'est pas nécessaire de créer un nouvel exemplaire de la partie dynamique (voir ch. II.4.1, §2.2). Il suffit d'initialiser la partie statique de façon à disposer d'un accès dynamique à toutes les représentations des valeurs constituant la partie dynamique de la nouvelle valeur.

Dans certaines constructions, la partie dynamique de la représentation n'existe pas.

Exemples :

1) modification élargir transformant une valeur de mode *bits* en un tableau de booléens.
2) "collatéral" non neutre construisant une valeur de mode tableau à partir de valeurs indépendantes.

Il est alors nécessaire de créer cette partie dynamique et de l'initialiser. Sa taille est toujours connue statiquement,ce qui permet de l'allouer dans la zone d'évaluation de l'environnement courant.

L'initialisation conduit à produire le code copiant les éléments du tableau.

Chaque fois que la représentation de la valeur n'existe pas en totalité, nous venons de voir qu'il était nécessaire de créer la partie statique dans la zone d'évaluation locale. Le descripteur que nous construisons, pour la valeur finalement retournée, va donc décrire cet emplacement.

Exemple : Considérons le programme suivant :

début
  [*10:20, 3:9*] *ent* *t* = ...
  ...
...*t*[*12:18, 4:7*]
  ...
fin

Si d_{trt} est le déplacement de l'emplacement réservé dans la zone d'évaluation pour la partie statique de la tranche, le descripteur δtrt décrivant la valeur finale est :

$\delta trt = (C_{rre}, (d_{trt}$ |))

| → δbase de l'environnement courant

C_{rre} représente les caractéristiques d'une valeur de mode [,] *ent*

2.3 - Conclusion -

Nous venons d'examiner l'ensemble des cas possibles permettant de construire en Algol 68 une valeur ; pour chaque cas nous avons montré comment l'on obtient le descripteur synthétisé à partir de l'arbre abstrait. Apportons une précision sur la structure des descripteurs de valeur :

les valeurs sont décrites pendant la génération de code par une chaîne de descripteurs et non par un seul descripteur.

Nous continuons cependant dans la suite à parler de descripteur et non de chaîne de descripteurs pour des raisons de commodité.

Sur cette chaîne, dont chaque élément représente soit un objet Algol 68, soit une notation, un certain nombre de descripteurs servent en fait de base aux descripteurs les précédant dans la chaîne.

Par extension, nous représentons, dans la suite, toutes les bases par des descripteurs (pas seulement celles associées à un objet Algol 68). Dans le cas de la base courante de pile par exemple ou de la base de la table des constantes l'information mode n'a plus de sens.

La construction de cette chaîne de descripteurs permet de synthétiser sans produire de code l'information concernant l'accès. Le parcours de cette chaîne (qui entraîne la génération du code réalisant l'accès à l'exécution) n'est fait que lorsque cet accès est nécessaire.

3 - SEMANTIQUE DES ENVIRONNEMENTS -

3.1 - Définition -

Cet aspect de la sémantique regroupe toutes les notions introduites par les changements d'environnements se produisant pendant l'exécution d'un programme. Ces créations et suppressions d'environnements sont spécifiques de la structure du langage et permettent à l'utilisateur de préciser la constitution et la gestion du cadre dans lequel doit se dérouler toute action algorithmique de son programme. Ces actions sont associées à chaque noeud du "squelette sémantique". Elles manipulent des valeurs et peuvent modifier la structure globale de l'ensemble des environnements existants.

Exemple :

- l'appel d'un texte de routine entraîne la création d'un nouvel environnement.
- une instruction de branchement peut avoir comme conséquence la suppression de plusieurs environnements.

Depuis les premiers langages algorithmiques (Fortran) où la notion est esquissée et surtout depuis les langages à structure de blocs, cette notion d'environnement à l'exécution est devenue familière aux utilisateurs bien qu'ils n'aient pas toujours conscience des traitements statiques ou dynamiques qui lui sont associés.

3.2 - Accès aux environnements actifs -

L'accès aux environnements actifs, permettant l'accès aux objets non locaux à l'environnement courant, est l'aspect le plus important de la sémantique des environnements.

Nous avons vu qu'à l'exécution (voir ch. II.4.1, §2.3.2), cet accès se fait par l'intermédiaire d'un vecteur contenant les bases des environnements actifs (display). Ce vecteur se trouve dans la zone statique associée à chaque texte de routine.

Chaque élément de ce vecteur contient une valeur pouvant servir de base pour l'accès à un objet non local.

Pour pouvoir construire un descripteur pour les objets non locaux pendant la génération de code il est nécessaire de donner une représentation pour chacune de ces bases. La représentation la plus naturelle est un descripteur qui ne comporte que les informations décrivant l'accès à l'environnement correspondant.

Pour la compilation de chaque texte de routine on crée un ensemble ordonné de descripteurs décrivant les éléments de ce vecteur. Cet ensemble est ordonné suivant les niveaux statiques croissants des environnements auxquels les descripteurs se rapportent.

Ces descripteurs sont utilisés pour construire ceux qui représentent des objets non locaux (par exemple sur le noeud (idf)).

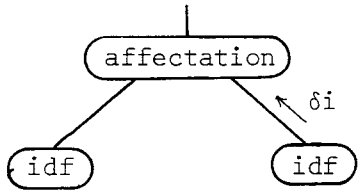
A la fin de la compilation d'un texte de routine on restaure l'ensemble des descripteurs d'environnements du texte de routine englobant.

Remarque :
La décoration identificateur doit contenir une information associée au niveau statique de l'identificateur pour que l'on puisse déterminer le descripteur de la base à utiliser.

Exemple : Soit la séquence suivante :

```
début
   ent i = 2,
   proc p := ent :
      début  ent j ;
             j := i ;
             j
      fin ;
   p
fin
```

Si l'on considère le sous-arbre associé à l'affectation j := i



le descripteur décrivant la valeur de la partie droite est schématisé par :

$\delta i = (C_{ent}, (d_i,$. $))$

. → base de l'environnement du programme principal

d_i est le déplacement de la valeur possédée par i dans la zone des variables de l'environnement du programme principal.

Si depdis1 est le déplacement -dans la zone statique de l'environnement courant- de l'élément (du vecteur contenant les bases d'environnements actifs) contenant la base de l'environnement de déclaration de i (environnement du programme principal), le descripteur δbasprog décrivant cette base s'écrit :

$\delta basprog = (C_{base}, (depdis1,$. $))$

. → δbase de l'environnement courant

C_{base} représente les caractéristiques d'une base.

Dans ces conditions, nous avons la configuration :

$\delta i = (C_{ent}, (d_i$. $))$

. → $\delta basprog = (C_{base}, (depdis1,$. $))$

. → δbase de l'environnement courant

3.3 - Conclusion -

Il est possible de donner une définition beaucoup plus étendue de la sémantique des environnements. Par exemple on peut considérer l'opération d'affectation comme faisant partie de cette sémantique en décrivant son action par :

environnement avant ——— affectation ——— environnement après

Ceci revient à décrire l'affectation par l'"influence" qu'a cette opération sur l'environnement. Il s'agit là d'une approche plus approfondie (que celle que nous avons faite) de la "sémantique opérationnelle", caractéristique des méthodes impératives et algorithmiques.

II.4.3 - SCHEMA DE GENERATION DE CODE -

1 - LES DIFFERENTES PHASES DE LA TRADUCTION -

Nous avons séparé le processus de production de code, pour un calculateur donné, en deux niveaux :

a)la génération d'un code intermédiaire sous forme de "quadruplets". Ces quadruplets sont produits pendant le parcours de l'arbre abstrait, squelette du programme à compiler, en fonction des règles sémantiques du langage, du système à l'exécution et des caractéristiques générales de la "famille" dont fait partie le calculateur qui supporte l'exécution du programme compilé. Les particularités technologiques de ce calculateur n'intervenant pas dans cette phase, la portabilité du code intermédiaire est préservée pour les calculateurs de la même famille.

Un quadruplet se représente par :

(code opération, opérande1, opérande2, opérande3).

Les opérandes sont des descripteurs construits pendant le parcours de l'arbre abstrait ou des valeurs immédiates.

b)la génération du code adapté au calculateur choisi.
Cette production est réalisée par traduction des quadruplets produits par la première phase. Cette traduction se fait en deux étapes distinctes correspondant à deux programmes traducteurs :

- le premier traducteur travaille au coup par coup et produit du code chargeable. Cette traduction est entreprise lorsque l'ensemble de quadruplets, correspondant à une "action sémantique de base", a été engendré par la phase précédente. Ces "actions sémantiques" correspondent aux "points de génération" indiqués par les paragraphes du Rapport traitant de la sémantique du langage. A ce point, les caractéristiques technologiques du calculateur (registres, code condition ...etc...) deviennent prédominantes puisqu'il s'agit de produire du code, accédant à des valeurs décrites par des chaînes de descripteurs et manipulant ces valeurs. Ce traducteur termine le processus de compilation et délivre ce que l'on appelle communément le "programme objet".

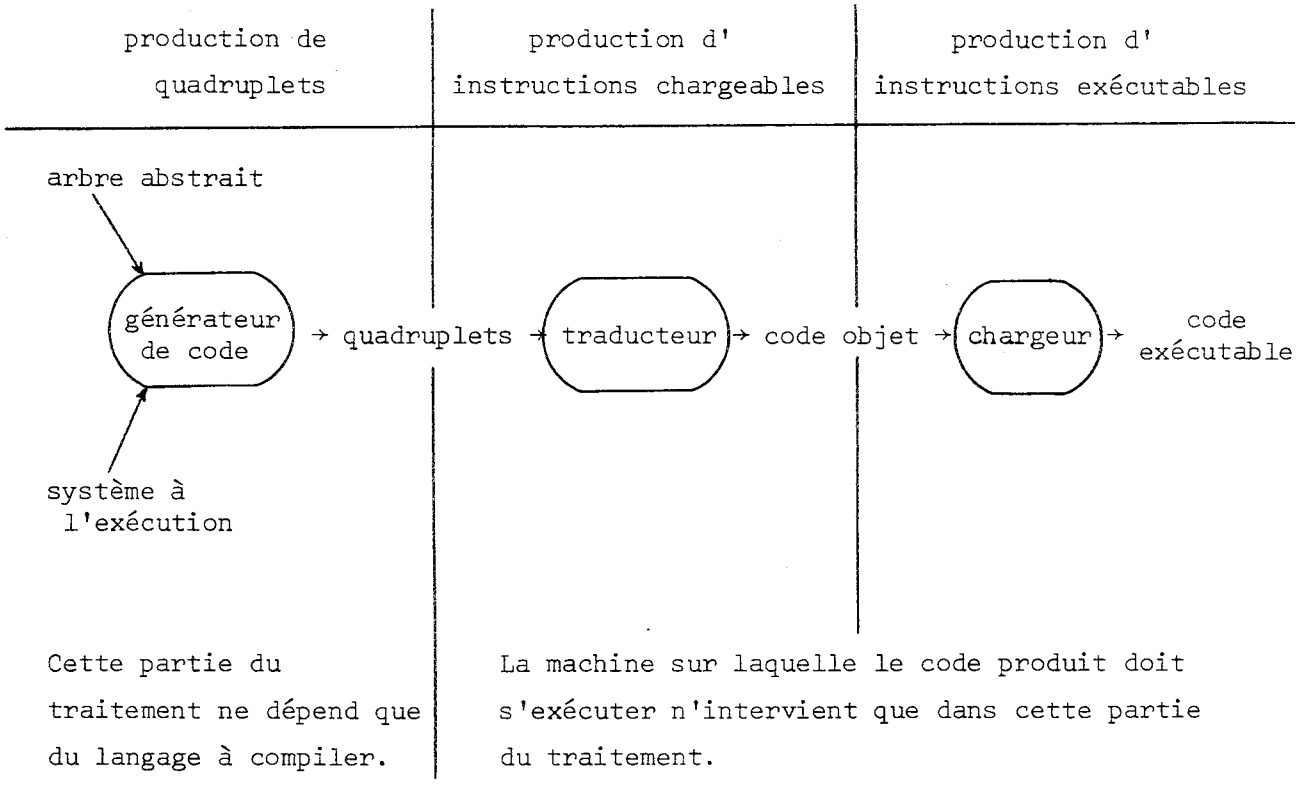
- le deuxième traducteur est un programme chargeur travaillant sur l'ensemble du code, produit par le traducteur précédent, pour un programme source donné. Dans notre cas, ce chargeur présente quelques particularités. En effet, indépendamment de ses fonctions de chargement, il réalise des traitements spécifiques de notre méthodologie ou du calculateur.
Citons par exemple :

- la prise en compte des options de compilation indiquées par le programmeur. Pour un programme source donné, le programme objet résultant du processus de compilation contient toutes les vérifications que l'utilisateur peut demander aux moyens des options. Le code correspondant à chaque vérification est encadré d'indicateurs le caractérisant. Chaque fois que le chargeur rencontre un tel indicateur, il décide de charger ou non le code de la vérification, suivant que l'option correspondante est ou non précisée.

- la résolution des problèmes posés par la limitation des déplacements dans les instructions machine d'un calculateur IBM 360. Dans ce calculateur les déplacements, dans les formats des instructions, ne peuvent être supérieurs à 4096 (4K). Nous n'avons pas voulu faire intervenir cette contrainte dans la production du programme objet et nous avons reporté son traitement au niveau du chargeur.

En définitive, le schéma complet de production de code est le suivant :
la génération, à partir de l'arbre abstrait, construit pour chaque noeud une séquence de quadruplets qui sont transformés, au fur et à mesure de leur production, en code objet.

Au niveau d'un noeud nous avons le schéma :



La description, que nous allons introduire et qui est développée dans les chapitres suivants, concerne la fonction de traduction de l'arbre abstrait en quadruplets. Nous n'y précisons pas la transformation d'un quadruplet en code exécutable, les traducteurs correspondants étant trop liés au calculateur pour lequel le code est produit. Ce problème est envisagé dans la troisième partie, en prenant, à titre de support d'exemples, le calculateur IBM 360.

2 - CONSTRUCTION D'UNE SEQUENCE DE QUADRUPLETS A PARTIR DE L'ARBRE ABSTRAIT -

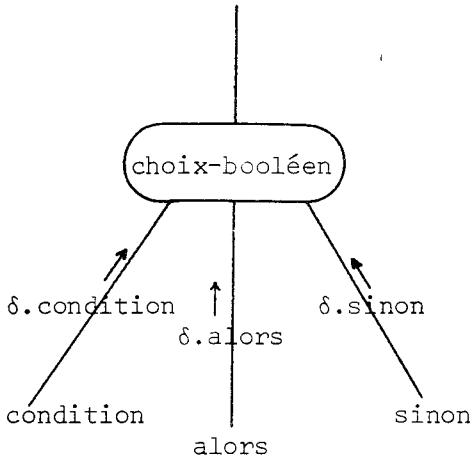
2.1 - Les modèles -

Considérons l'arbre abstrait tel que nous l'avons défini dans les chapitres précédents.

Nous avons indiqué comment s'effectue le parcours de cet arbre et de quelle manière sont synthétisés les descripteurs de valeur. Indépendamment de ces descripteurs et des décorations attachées aux noeuds, d'autres informations sont héritées ou synthétisées pendant le parcours de l'arbre.

A chaque noeud de l'arbre abstrait, on peut associer une ou plusieurs "actions de génération" correspondant aux traitements sémantiques associés à ce noeud.

Exemple : Considérons le noeud (choix-booléen) et ses opérandes.
Le traitement correspondant peut s'écrire de manière informelle :



1 Traitement du sous-arbre de la condition avec synthèse de δ.condition

2 Génération du test de la condition
Génération d'une instruction de branchement-si-faux à l'étiquette E1

3 Traitement du sous-arbre de la partie *alors* avec synthèse de δ.alors

4 Génération de la mise en ressource-résultat de la valeur décrite par δ.alors
Génération d'une instruction de branchement inconditionnel à l'étiquette E2
Génération de la définition de l'étiquette E1

5 Traitement du sous-arbre de la partie *sinon* avec synthèse de δ.sinon

6 Génération de la mise en ressource-résultat de la valeur décrite par δ.sinon
Génération de la définition de l'étiquette E2

Les trois parties encadrées correspondent aux trois actions de génération spécifiques du choix booléen. Remarquons que chacune de ces actions n'est entreprise qu'après synthèse du descripteur de la valeur à manipuler.

Pour deux instructions de choix booléen différentes mais délivrant une valeur de même mode, les séquences de quadruplets correspondant à chacune de ces actions ne vont différer que par les opérandes y intervenant. Par contre, les séquences d'instructions des programmes objets correspondants vont différer suivant les emplacements des représentations des valeurs manipulées.

En conséquence, à chaque action de génération nous pouvons associer un modèle de la séquence des quadruplets à produire pour cette action. Lorsque nous disposons de l'ensemble des opérandes effectifs (arguments) correspondant aux opérandes d'un modèle, il suffit d'"activer" ce modèle pour engendrer la séquence de quadruplets associés.

Un modèle est donc une séquence non activée de quadruplets dont les opérandes dépendent des mêmes informations contextuelles. C'est l'activation qui transmet à un modèle les informations qui lui sont nécessaires. Cette transmission s'opère par une identité entre les opérandes (formels) et les arguments (effectifs).

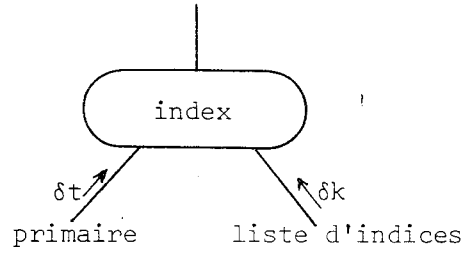
L'écriture des modèles, devant tenir compte de la sémantique du langage et du système à l'exécution, n'est guère automatisable avec les méthodes actuelles de définition des langages.

2.2 - Exemple -

Au cours de la génération de code pour le noeud (index), on est amené à produire le code qui vérifie si un indice est compris entre les bornes bi et bs de la dimension du tableau auquel cet indice est associé.

Soit le programme et l'arbre abstrait :

```
début
   ent i,j,k ;
    :
   [i:j] ent t ;
    :
   t[k] := ...
    :
fin
```



Le traitement de l'arbre du primaire synthétise le descripteur δt à partir duquel nous construisons les descripteurs δtbi et δtbs décrivant les bornes du tableau t.

Le traitement de la liste d'indices synthétise le descripteur δk (il n'y a qu'un indice) décrivant l'indice.

La génération de la vérification fournit la séquence de quadruplets :

(comparer, arith, δk, δtbi)

co comparaison arithmétique entre k et la borne inférieure de t co

(branchement, < , , biblioerreurbinf)

co génération d'un appel à une routine de bibliothèque de traitement d'erreur si l'indice est inférieur co

(comparer, arith, δk, δtbs)

co comparaison arithmétique entre k et la borne supérieure de t co

(branchement, > , , biblioerreurbsup)

co génération d'un appel à une routine de bibliothèque de traitement d'erreur si l'indice est supérieur co

Pour obtenir ces quadruplets nous activons le modèle *vérifindice* dès que nous disposons de δk, δtbi et δtbs

modèle vérifindice = (descr δindice, δbi, δbs)

co ce modèle a trois opérandes qui sont les descripteurs de l'indice, de la borne inférieure et de la borne supérieure

co

(comparer, arith, δk, δbi)

(branchement, <, , biblioerreurbinf)

(comparer, arith, δk, δbs)

(branchement, >, , biblioerreurbsup)

finmodèle

II.4.4 - LE CALCULATEUR -

1 - FONCTIONS ET CARACTERISTIQUES -

Le calculateur sert à l'exécution du code produit.

Ce calculateur est caractérisé dans la description par :

- un jeu d'instructions permettant de réaliser toute opération utilisée dans les quadruplets (voir ch.II.4.5, § 6). La traduction d'un quadruplet en code machine dépend de ce jeu d'instructions.

- des ressources mémoire utilisées dans la mise en oeuvre du système à l'exécution (voir ch.II.4.1, §2).

Un calculateur réel comporte des ressources mémoire ayant des caractéristiques différentes (mémoire centrale, registres, accumulateurs, ...). Dans la description, nous évitons souvent de tenir compte de ces différences pour rester adaptés à une large classe de calculateurs. Dans la pratique nous utilisons au maximum les ressources d'accès et de calcul rapide dont dispose le calculateur. Ces ressources sont en nombre limité et leur utilisation met en oeuvre des techniques particulières.

2 - UTILISATION DES RESSOURCES DU CALCULATEUR -

Les calculateurs actuels possèdent des ressources (appelées : registres, accumulateurs, ...etc...) privilégiées pouvant être différenciées d'après leur emploi en :

- ressources d'adressage (apparaissant dans la partie adresse des instructions)
- ressources de calcul (apparaissant comme opérande dans les instructions).

Un écrivain de compilateur doit tenir compte de ces ressources afin de les utiliser au mieux dans le code produit car, indépendamment de la nécessité de les utiliser dans presque toutes les instructions-machine, elles ont l'avantage d'être accessibles beaucoup plus efficacement que les cellules-mémoire.
Ceci conduit à spécialiser certaines ressources d'adressage pour baser les constituants fondamentaux (c'est-à-dire les plus utilisés) du système à l'exécution (base de l'environnement courant, base de l'environnement global, base de la table des constantes ...etc...).

Pendant l'exécution, la mise à jour de ces ressources a lieu (si besoin est) pendant les changements d'environnement. Durant la phase de génération de code, des descripteurs d'un type particulier décrivent ces ressources (ces descripteu sont constants puisque -fonctionnellement- les ressources basent toujours les mêmes constituants).

Nous pourrions ne pas faire apparaître les ressources non spécialisées et les ressources de calcul dans la description du processus de génération de code. Ce ne serait pas pour éviter les problèmes que leur utilisation pose, mais parce qu'elles seraient considérées comme ne faisant partie que des caractéristiques techniques du calculateur. A ce titre, leur utilisation et leur gestion seraient prises en charge, dans la méthodologie que nous proposons, par le traducteur des quadruplets en code chargeable.

De cette façon, dans la description du traitement associé à tout noeud de l'arbre abstrait, le code produit (même s'il utilisait explicitement certaines de ces ressources, comme dans le noeud (index)), délivrerait finalement une valeur dont la représentation (accessible statiquement) serait toujours en mémoire.

Le travail du traducteur (quadruplet ⟶ code chargeable) consisterait alors à produire le code réalisant successivement :

a) l'accès à des représentations de valeurs. Chaque accès est décrit par une chaîne de descripteurs. Le traducteur va demander des ressources d'adressage si l'accès ne peut être codé directement dans la partie-adresse des instructions

b) les calculs à faire sur ces valeurs. C'est le type du quadruplet qui indique les opérations à effectuer. Les ressources de calcul, nécessitées par les instructions à engendrer, sont demandées par le codeur.

c) le rangement en mémoire de la représentation de la valeur obtenue. Cette opération peut aussi nécessiter des ressources d'adressage.

Au fur et à mesure de la production des instructions, le traducteur libèrerait (rendrait disponible) les ressources dès qu'elles ne lui seraient plus nécessaires.

Une telle méthode, bien qu'elle soit intéressante par les garanties de sécurité qu'elle apporte sur la gestion des valeurs, ne satisfait pas pleinement un écrivain de compilateurs.

En effet, si les problèmes d'adressage sont résolus de manière élégante et efficace par l'emploi des chaînes de descripteurs, un code plus performant peut être obtenu si les valeurs, que le traducteur manipule, sont le plus souvent possible conservées dans ces ressources privilégiées. Ceci revient à minimiser le code produit dans les phases a) et c) du travail du traducteur.

De ce fait, ces ressources, qui peuvent contenir -à l'exécution- les (accès aux) représentations des valeurs délivrées par le code produit pour certains noeuds de l'arbre abstrait, doivent être représentées par des descripteurs durant la phase de génération. Ces descripteurs ont un format analogue à celui des descripteurs associés aux ressources d'adressage spécialisées.
Se servir de ces ressources de façon non temporaire, nécessite, du fait de leur nombre généralement très limité, une gestion élaborée afin de produire le code utilisant les informations significatives qu'elles contiennent.

Sans entrer dans les détails de cette gestion indiquons en les deux principaux aspects.

2.1 - Elle doit assurer la durée de vie des informations significatives se trouvant dans les ressources -

Supposons qu'à un instant quelconque de l'exécution d'un programme, toutes ces ressources contiennent des informations significatives. Si la génération de l'instruction suivante nécessite des ressources disponibles, il faut décider la libération de certaines d'entre elles, pour continuer le traitement.
Les informations contenues dans les ressources choisies doivent obligatoirement être récupérées et mises ailleurs (puisqu'elles sont significatives). Du code est donc produit pour les sauvegarder dans la zone d'évaluation, ce qui permet de garder l'accès statique à ces informations.

Ces ressources forment donc un ensemble limité d'emplacements réservés aux résultats d'évaluations. A cet ensemble est adjoint la zone d'évaluation , comme "zone de débordement".

2.2 - Elle doit assurer statiquement le contrôle permanent de l'état global des ressources -

Lorsque l'ordre d'exécution des instructions d'un programme est identique à celui dans lequel elles ont été produites, le contrôle est réalisé automatiquemen
Le cas contraire se résume à deux situations :

2.2.1 - Contrôle de l'ordre d'exécution -

Il n'est généralement pas possible de déterminer statiquement l'ordre d'exécutio des instructions d'un programme. En effet, certaines constructions du langage réalisent des choix d'alternance et des déroutements (sauts).

- Choix d'alternance

Les instructions choix (choix booléen, choix entier, choix union) permettent à l'utilisateur de diversifier (en fonction de certains critères) les traitements qu'il veut réaliser. Le choix du traitement finalement exécuté étant fait dynamiquement (en testant la valeur fournie par la partie de contrôle de la construction correspondante), il faut engendrer le code de chaque alternance et les instructions de test et de branchement associées .

- Déroutements

Les instructions de branchement permettent à l'utilisateur de dérouter l'exécuti de son programme vers le code correspondant à l'unité étiquetée.

Si une phrase sérielle contient une ou plusieurs déclarations d'étiquettes (elle ne peuvent pas être déclarées dans une autre construction) l'ordre d'exécution de ses unités constituantes ne peut plus être déterminé statiquement.

En plus des instructions choix, les constructions, dont au moins un composant est une série, sont les instructions *bloc* (noeud (début)) et les instructions *pour* (noeud (itération)). Il est intéressant de savoir si les séries contenues dans ces instructions contiennent des déclarations d'étiquettes.

Lors de la génération de code pour ces deux types de construction il faut conserver le contrôle statique des ressources de calcul et d'adressage.

En commençant le traitement des noeuds correspondants nous connaissons statiquement quel sera, à l'exécution, l'état des ressources en ce point. Il est de même absolument nécessaire de connaître statiquement cet état après l'exécution des instructions (correspondant à chacune de ces constructions), pour pouvoir poursuivre le processus de génération. Le choix des instructions à exécuter étant fait dynamiquement, l'état final des ressources doit être indépendant de ce choix (c'est à-dire, de l'alternance choisie dans les instructions choix, des unités exécutées dans les séries).

D'autre part l'évaluation d'une telle construction fournit une valeur (sauf l'instruction d'itération) pouvant être utilisée. Dans ces conditions le code produit pour l'utilisation de cette valeur doit pouvoir la "récupérer" quel que soit le chemin parcouru.

Une solution à ces problèmes consiste à produire -en début de traitement de ces constructions- du code sauvegardant (dans la zone d'évaluation) les informations contenues dans toutes les ressources (de calcul et d'adressage non spécialisées) significatives. Les ressources sont alors toutes disponibles pour le code correspondant à ces constructions. La valeur résultat devant toujours être accessible de la même manière, le code terminant chaque alternance charge toujours (l'accès à) sa représentation dans la même ressource de calcul (ressource-résultat). De cette manière, seule la ressource-résultat est significative une fois la construction évaluée (les autres ressources utilisées dans la construction ne sont plus significatives). Pour poursuivre la génération de code il suffit de faire représenter, par le descripteur associé à la ressource-résultat, une valeur ayant les caractéristiques statiques (mode, taille ...etc...) de la valeur retournée.

Remarques :

1) Les *acheveurs* (*exit*) permettent aussi d'avoir plusieurs alternances délivrant des valeurs utilisables. Ces constructions utilisant explicitement une étiquette et étant contenues obligatoirement dans une série, la sauvegarde des ressources est produite lors du traitement des noeuds ayant comme opérande la série en question. Il suffit donc de produire la mise en ressource-résultat de la représentation de la valeur fournie par chaque alternance.

2) Il n'est pas nécessaire lors du traitement des noeuds (début) d'engendrer le chargement en ressource-résultat. En effet, pour qu'au niveau de cette construction il puisse y avoir, à l'exécution, plusieurs alternances délivrant une valeur (utilisée depuis le même point du programme), il faut que la série constituante contienne des acheveurs. Dans ce cas, d'après la remarque précédente, les mises en ressource-résultat sont faites par le code associé aux acheveurs. Ceci n'est correct que si les zones des variables de deux blocs (disjoints) en position collatérale ne se recouvrent pas (voir le traitement du noeud (début) ch.II.4.6 §2).

2.2.2 - Appels -

La situation est dans ce cas totalement différente. Il ne s'agit plus de plusieurs alternances délivrant une valeur mais de plusieurs déroutements vers la même séquence de code (après avoir transmis les paramètres) avec retour au point d'appel et transmission de la valeur résultat, une fois ce code exécuté

Pendant la phase de génération, nous connaissons, pour chaque appel, l'état correspondant des ressources. Il n'y a pas de raison pour que cet état soit le même au début du traitement de chaque appel à une même routine. Or la génération du code pour la routine nécessite la connaissance statique d'un état des ressources. Cet état doit être commun à tous les points d'appel. De plus, il fau aussi transmettre la valeur-résultat. A cause du déroutement, sa représentation doit, à l'exécution, être accessible de manière unique depuis tous les points d'appel.
Nous appliquons la même solution que dans le cas précédent. Dans le traitement de chaque point d'appel, nous produisons le code sauvegardant, dans la zone d'évaluation, les informations contenues dans les ressources significatives. De cette manière, les ressources ne sont plus significatives et, donc, toutes disponibles lors du début de l'exécution de la routine.

Comme dans le cas précédent, (l'accès à) la représentation de la valeur-résultat peut être chargée, par le code de la routine, dans la ressource-résultat. En retour, cette ressource est la seule significative puisque celles utilisées par le code de la routine ont été libérées dès qu'elles n'étaient plus significative Pour poursuivre la génération au niveau d'un appel, il suffit d'indiquer, dans le descripteur associé à la ressource-résultat, les caractéristiques de la valeur retournée.

Remarques :

1) La transmission de la valeur résultat (si elle existe) se fait avec changement d'environnement et nous retrouvons les problèmes d'accessibilité à la représentation de la partie dynamique (si elle existe) de la valeur.

   La solution que nous avons indiquée dans la description de la fonction de traduction assure cet accès. On peut lui faire le reproche de n'être pas très performante mais il est toujours possible de l'améliorer. Par exemple, on peut différer le traitement (création, dans l'environnement courant, d'un nouvel exemplaire des parties dynamiques de la valeur si celle-ci était locale à la routine) afin de ne le faire que lorsque la représentation risque d'être écrasée, c'est-à-dire avant toute allocation d'espace.

Ceci augmente les performances car en pratique la valeur est très souvent utilisée (et donc n'est plus à sauvegarder) avant d'avoir à allouer de l'espace.

2) Choisir de créer un environnement non par région (bloc) mais uniquement par routine, limite la production du code, assurant l'accessibilité des valeurs-résultat, uniquement au cas des valeurs retournées par les routines (c'est-à-dire au niveau des noeuds (appel)).

Après cette longue introduction à la génération de code, qui avait pour but de rappeler ou d'introduire les notions nécessaires à une bonne compréhension des mécanismes que nous utilisons, nous allons, dans les chapitres suivants, décrire en détail la phase de production des quadruplets à partir de l'arbre abstrait décoré.

L'utilisation de quadruplets et de modèles dans un processus de génération de code permet de faire abstraction de beaucoup de contraintes imposées par le calculateur et d'obtenir un code intermédiaire plus aisément portable. C'est le choix de ces quadruplets qui fixe le degré de la portabilité obtenue.

D'autre part, dans les procédés d'héritage et de synthèse (notamment des descripteurs de valeurs) nous trouvons un moyen bien adapté pour assurer certains aspects de la localité de la traduction. Ceci rend la description plus claire car les informations nécessaires pour le traitement associé à un noeud ne sont fournies que par les noeuds qui lui sont adjacents.

II.4.5 - NOTATIONS ET CONVENTIONS UTILISEES DANS LA DESCRIPTION -

1 - INTRODUCTION -

Dans cette description nous n'utilisons pas le formalisme des attributs sur une arborescence comme pour le processus de décoration. Nous reviendrons sur ce choix au chapitre II.5.

Le langage de description présenté précédemment est mieux adapté à notre problème. Il met notamment en évidence les élaborations séquentielles ou collatérales, et offre des possibilités de descriptions plus "réalistes" que les attributs. Nous montrons dans la suite, qu'une implantation raisonnable peut être obtenue à partir d'une telle description. Il n'est pas encore possible d'avoir la "meilleure" implantation (en termes d'efficacité du code produit) pour un problème donné ; cependant les résultats obtenus laissent espérer que de telles méthodes finiront par donner une solution permettant d'aboutir à une implantation propre et efficace.

2 - OBJETS UTILISES DANS LA DESCRIPTION -

2.1 - Caractéristiques -

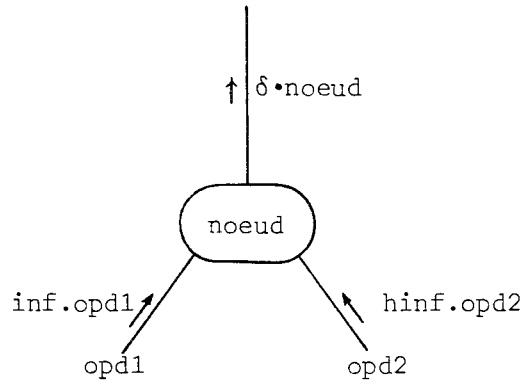
Rappelons les caractéristiques des principaux objets intervenant dans la description (voir ch. II.1, §.2.1).

- Les valeurs à l'exécution sont représentées par des descripteurs de mode descr.
- Les modes Algol 68 sont des objets de mode type.
- Les noeuds possèdent des décorations. Rappelons que l'utilisation de la décoration déco du noeud N est notée déco de (N).

Le mode des décorations fondamentales a été donné précédemment. Celui des autres est précisé au niveau de chaque noeud.

- Un sous-arbre est un objet de mode arbre.
- Dans la description, ces objets peuvent être alloués localement au moyen d'une déclaration (par exemple : descr descripteur). Ils peuvent aussi être transmis en paramètres hérités ou synthétisés ; ils sont alors notés comme des attributs sur un arbre définis au paragraphe 1.2 du chapitre II.1.

Exemple :



Notations pour les descripteurs :

Ils sont toujours préfixés par le symbole δ.
La flèche associée à la branche indique s'ils sont hérités ou synthétisés.
Le sélecteur de la branche figure après un point.

On dispose enfin de décorations de mode étiquette permettant de définir des étiquettes ou des points d'entrée.

2.2 - Objets globaux -

Un certain nombre d'objets globaux sont utilisés dans toute la description; il s'agit :

- de descripteurs de valeurs constantes,
- de descripteurs d'une zone de transfert d'information aux routines de bibliothèque
- de l'ensemble des descripteurs utilisés pour la sémantique des environnements.

2.2.1 - Routines de bibliothèque -

Pour éviter de produire plusieurs fois la même séquence de code correspondant à un traitement spécifique, on peut la transformer en routine de bibliothèque à l'exécution et ne produire que l'appel. On obtient ainsi un gain de place pour le code produit au détriment d'une certaine perte d'efficacité (rapidité) à l'exécution correspondant à l'appel et au passage d'informations entre ces routines de bibliothèque et le programme objet. Au niveau de la réalisation pratique on peut réduire par des optimisations locales cette perte d'efficacité. Dans la description, ces routines servent d'une part,à effectuer des traitements que l'on retrouve chaque fois que l'on manipule des valeurs dynamiques,et d'autre part,à gérer dynamiquement l'espace mémoire (pile ou tas) (voir ch. II.4.3, § 4.1).

Une ressource particulière (par exemple, des registres) est réservée à la transmission d'informations à ces routines. On dispose dans cette ressource de plusieurs zones représentées par les descripteurs :

δparambibliothèque
δparambisbibliothèque
δparamterbibliothèque
δrésultbibliothèque

2.2.2 - Ensemble de descripteurs représentant le vecteur d'accès aux bases des environnements actifs -

Cet ensemble de descripteurs, noté Δenvcour , est continuellement accessible pendant la génération du code associé à n'importe quel texte de routine. Il est mis à jour en début et en fin du traitement de chaque texte de routine. Le descripteur δtêtepile, appartenant à cet ensemble, décrit l'accès au sommet de la zone dynamique.

3 - FONCTIONS UTILISEES DANS LA DESCRIPTION -

Un certain nombre de fonctions opèrent sur les objets définis au paragraphe précédent et permettent, soit de les initialiser partiellement ou totalement, soit de sélectionner des informations qu'ils contiennent.

Nous donnons les spécifications des fonctions utilisées dans toute la description. D'autres fonctions sont définies localement avec la description de chaque noeud, elles se rattachent à l'une ou l'autre des catégories que nous envisageons ici.

3.1 - Fonctions associées aux modes des valeurs Algol 68 -

Il s'agit des fonctions suivantes :

- taille,
- dynamique.

```
fonction taille = (type mode) ent
    co
      calculer la taille de la partie statique du mode transmis en paramètre.
      Cette taille est transmise comme résultat.
    co
finfonction

fonction dynamique = (type mode) bool
    co
      délivrer la valeur "vrai" ou "faux" suivant que le mode a des parties
      dynamiques effectives ou pas
    co
finfonction
```

3.2 - Fonctions associées aux descripteurs -

Nous définissons quatre types de fonctions associées aux descripteurs.

3.2.1 - Des fonctions permettent d'accéder aux informations d'un descripteur ou de les modifier : complétermode et modedescr.

fonction complétermode = (descr δ, type mode) descr
    co
      compléter le descripteur δ pour qu'il représente un objet du mode demandé. Le descripteur δ, ainsi modifié est retourné comme résultat
    co
finfonction

fonction modedescr = (descr δ) type
    co
      faire délivrer comme résultat le mode de la valeur représentée par le descripteur δ.
    co
finfonction

3.2.2 - Des fonctions créent un nouveau descripteur.

- Le descripteur est créé à partir d'un autre dont on veut modifier l'accès :

  modifieraccès.

- Le descripteur est créé à partir d'une décoration :

  identificateur : descridf
  indicateur : descrindic
  opérateur : descrop.

fonction modifieraccès = (ent dépla, descr δ) descr
    co
      allouer et initialiser un descripteur à partir de δ en modifiant l'accès à l'objet qu'il représente, d'une quantité égale à dépla. Le descripteur créé est retourné comme résultat.
    co
finfonction

fonction descridf = (*identificateur* i) *descr*

co

allouer et initialiser un descripteur pour qu'il représente la valeur possédée par l'objet décrit par *identificateur* i. Le descripteur, une fois initialisé , est retourné comme résultat.

co

finfonction

fonction descrindic = (*indicateur* indic) *descr*

co

allouer et initialiser un descripteur pour qu'il représente l'objet associé à l'indicateur précisé par indic . Le descripteur, une fois initialisé, est retourné comme résultat.

co

finfonction

fonction descrop = (*opérateur* op) *descr*

co

allouer et initialiser un descripteur pour qu'il représente la valeur possédée par l'objet décrit par *opérateur* op. Le descripteur, une fois initialisé, est retourné comme résultat.

co

finfonction

- Le descripteur est créé à partir d'un descripteur de partie statique de tableau pour décrire l'un quelconque de ses champs.

fonction orivirt = (*descr* δtab) *descr*
    co
      Création d'un descripteur δ à partir du descripteur δtab.
      δ représente l'origine virtuelle se trouvant à l'exécution dans la partie statique du tableau décrit par δtab.
    co
finfonction

fonction bi = (*int* k, *descr* δtab) *descr*
    co
      Création d'un descripteur δ à partir du descripteur δtab.
      δ représente la borne inférieure de la $k^{ième}$ *dimension du tableau décrit par δtab. Cette borne est, à l'exécution, dans la partie statique du tableau.*
    co
finfonction

fonction bs = (*int* k, *descr* δtab) *descr* .
    co voir bi co
finfonction

fonction e = (*int* k, *descr* δtab) *descr*
    co
      Création d'un descripteur δ à partir du descripteur δtab.
      δ représente l'enjambée de la $k^{ième}$ *dimension du tableau décrit par δtab. Cette enjambée est, à l'exécution, dans la partie statique du tableau.*
    co
finfonction

3.2.3 - Des fonctions sont associées à la gestion statique des ressources.

Elles allouent une ressource et créent un descripteur dont l'accès décrit l'emplacement alloué. Nous définissons ainsi :

pour la zone d'évaluation : allocpse,
pour la table des constantes (constituée pendant la génération de code) : constante, fairefantome.

Une fonction d'un type particulier décrit une notation de mode *proc* (un texte de routine) : faireroutine.

Une fonction permet de décrire une valeur en ressource-résultat : faireres.

```
fonction allocpse = (ent long) descr
    co
      réserver dans la zone d'évaluation un emplacement dont la taille est
      donnée par le paramètre long. Allouer et initialiser un descripteur
      pour qu'il permette d'accèder à cet emplacement. Le descripteur, ainsi
      crée, est retourné comme résultat.
    co
finfonction

fonction constante = (ent valeur) descr
    co
      ranger la valeur entière (il s'agit d'une valeur de compilation) dans
      la table des constantes. Allouer et initialiser un descripteur pour
      qu'il représente cette valeur en table. Ce descripteur est retourné
      comme résultat.
    co
finfonction

fonction fairefantome = (type mode) descr
    co
      construire et mettre en table des constantes une valeur fantome du
      mode demandé. Allouer et initialiser un descripteur pour qu'il
      représente cette valeur. Ce descripteur est retourné comme résultat.
    co
finfonction

fonction faireroutine = (descr δ) descr
    co
      Indiquer dans le descripteur qu'il représente un texte de routine.
      Le descripteur, ainsi complété, est retourné comme résultat.
    co
finfonction
```

fonction faireres = (descr δ) descr

co

faire représenter par δ une valeur en ressource-résultat. Le descripteur une fois complété est retourné comme résultat.

co

finfonction

3.2.4 - Deux fonctions gèrent un ensemble de descripteurs permettant de représenter les objets non locaux :

nouvelenviron, libérerenviron.

fonction nouvelenviron = (ent psi) ensemble descr

co

création d'un nouvel environnement de compilation.
psi : taille de la zone des variables. Elle sert à initialiser le calcul de la taille de la zone statique.
Cette fonction construit un ensemble de descripteurs représentant le nouvel environnement, initialise cet environnement et rend comme résultat cet ensemble de descripteurs.

co

finfonction

fonction libérerenviron = ensemble descr

co

libération de l'environnement de compilation.
Cette fonction récupère l'ensemble de descripteurs du précédent environnement de compilation et le retourne comme résultat.

co

finfonction

3.3 - Une fonction permet de tester si un sous-arbre est vide -

fonction nonnil = (arbre a) bool

co

Retourner la valeur "faux" ou "vrai" suivant que l'arbre a est vide ou ne l'est pas.

co

finfonction

4 - ACTIONS UTILISEES DANS LA DESCRIPTION -

A l'inverse des fonctions qui délivrent un résultat mais qui ne produisent pas de code, les actions ne délivrent pas de résultat mais peuvent produire du code, gérer et initialiser des ressources, réduire des sous-arbres. Elles ne sont introduites que pour simplifier et structurer la description suivant les techniques d'analyse descendante préconisées en programmation structurée [DIJKSTRA].

Comme pour les fonctions, nous ne précisons ici que les actions utilisées dans toute la description. Les actions propres à un noeud ou un ensemble de noeuds sont définies ultérieurement.

Nous définissons trois actions permettant de traiter les modifications, les copies de valeur (complètes ou partielles), les valeurs résultat.

4.1 - Les modifications -

Les modifications sont de six types : dérepérer, déprocédurer, unir, ranger, élargir, neutraliser. Au niveau de chaque noeud auquel des modifications peuvent s'appliquer on dispose d'une décoration modification, à appliquer au descripteur synthétisé sur le noeud.

```
action gmodif = (descrδ , modification modif)
    co δ : descripteur représentant la valeur à modifier
       modif : liste de modifications co
    descr δmodif
    tantque modif s'écrit mdf + modif
        faire
            co traitement de la modification co
            cas codemdf (mdf) dans
```

```
dérepérer :      co construit à partir de δ le descripteur δmodif représentant
                    la valeur dérepérée (voir sémantique des valeurs,
                    ch. II.4.2., § 2.2.3) co

déprocédurer :   co δ représente une valeur de mode procédure co
                 activerroutine(δ)
                    co activation de la routine co
                 construireappel(δmodif, résultat(modedescr(δ)))
                    co transfert la valeur dans l'environnement courant et
                    construit le descripteur δmodif la décrivant co

unir :           δmodif ← allocpse(taille(mode(mdf)))
                    co allocation d'espace dans la zone d'évaluation pour la
                    partie statique co
                    co génération de code pour initialiser la partie statique
                    de la valeur. La partie statique de l'union est formée
                    - d'un en-tête contenant l'identification du composant
                    - de la partie statique du composant co

ranger :         δmodif ← allocpse(taille(modemdf(mdf)))
                    co allocation d'espace dans la zone d'évaluation pour la
                    partie statique co
                 affectation(δmodif, δ, statique)
                    co initialisation de la partie statique co

élargir :        co allocation en zone d'évaluation d'un emplacement suscep-
                    tible de contenir la partie statique et dynamique éventuel-
                    le de la valeur co

                 co génération du code permettant d'initialiser cette nouvelle
                    valeur représentée par le descripteur δmodif co

                 fincas
                 δ ← δmodif
                 complétermode (δ, modemdf(mdf))
              finfaire
                    co fin du parcours de la liste de modifications; le descrip-
                    teur δ décrit la valeur après toutes ces modifications
                    statiques ou dynamiques co
finaction co fin de l'action gmodif décrivant le traitement des modifications
            pendant la génération de code co
```

Dans cette description nous avons utilisé les fonctions et actions suivantes :

action construireappel = (descr δ, type moderesultat)
co voir le traitement du noeud (appel) co
finaction

fonction modemdf = (modification mdf) type
co délivre le mode a posteriori associé à la modification co
finfonction

fonction codemdf = (modification mdf) code
co délivre le code de la modification co
finfonction

4.2 - Copies de valeurs -

Nous avons regroupé dans deux actions : affectation et copie dynamique l'ensemble des actions de copie de valeurs.

Au niveau de la génération, nous distinguons deux types d'action de copie.

4.2.1 - Les actions de copie de la partie statique d'une valeur -

L'espace nécessaire à la nouvelle partie statique est déjà alloué dans la zone statique, indépendamment de cette action ; son accès est décrit par un descripteur. La copie est faite globalement, en général par une seule instruction machine.

Exemples : - affectation d'une valeur entière
- affectation d'une valeur de mode repère de tableau
- copie des parties statiques des paramètres d'une procédure

4.2.2 - Les actions de copie des parties dynamiques d'une valeur -

Cette copie est indépendante de la précédente, en général. On dispose, pendant la génération de code, d'un descripteur décrivant la valeur (et notamment l'accès à sa partie statique). L'espace pour la partie dynamique n'est pas toujours alloué.

La copie nécessite la génération de code pour un parcours dynamique de la représentation de la valeur (voir ch. II.4.2, § 2.2.2). Ce parcours énumère séquentiellement les éléments dynamiques (tableaux, champs dynamiques d'une structure) composant la valeur pour lesquels la copie doit être faite. La stratégie adoptée est la suivante : on copie pour chaque élément les parties statiques et dynamiques avant de passer au suivant. On peut noter que les éléments peuvent également être dynamiques. Ces informations sont connues statiquement grâce au mode de la valeur, et l'on peut en conséquence, engendrer le code adéquat .

Certaines copies demandent une allocation préalable d'espace. Pour garder la structure de l'action de copie, éviter des difficultés éventuelles avec la récupération dynamique d'espace sur le tas, et éviter des parcours supplémentaires de la représentation de cette valeur, nous allouons cet espace par couches. (Une couche est formée de l'ensemble des parties statiques des éléments d'un tableau). Cette stratégie permet d'initialiser correctement la partie statique associée. On engendre ainsi le code pour construire une valeur arborescente (on construit les parties statiques associées à chaque niveau).

D'autres copies ont pour but, sous certaines conditions, de préserver l'accessibilité à une partie dynamique. Le code pour l'allocation d'espace et la copie doit être précédé (pour l'ensemble de la valeur) de la génération d'un test permettant de décider dynamiquement si cette partie dynamique est dans l'environnement actif courant. La représentation d'une valeur ne peut en effet chevaucher plusieurs environnements.

Remarques :

1) - La technique de recopie utilisée (élément par élément) nous amène dans le cas de l'affectation à reprendre ce que nous avions énoncé à propos de l'élaboration collatérale des valeurs (voir ch. II.4.1., § 2.2). L'élaboration d'une affectation conduit à élaborer collatéralement sa source et sa destination sans protection des valeurs obtenues. La copie ayant lieu élément par élément, on peut dans certains cas modifier la source.

Exemple : affectation de deux tranches (exemple fourni par le groupe de travail de Cambridge).

```
début
(1 : 5) ent t ;
...
t (2 : 5) := t (1 : 4)
fin
```

Si l'on ne protège pas la source par copie dans la pile, le résultat obtenu dépend de l'ordre dans lequel les éléments ont été affectés. Tous les cas où cette situation se produit, ne sont pas décelables statiquement. La détection dynamique consiste à comparer les adresses (d'implantation en mémoire) des représentations des deux tableaux. Si ces représentations se recouvrent, on protège la source par une copie supplémentaire.

2) - Les tableaux flexibles dont les représentations se trouvent sur le tas ne posent pas de problèmes particuliers dans la mesure où ils sont détectés statiquement d'après le mode de la valeur (dans les affectations par exemple). On peut alors produire le code permettant d'allouer de l'espace et d'effectuer la copie.

3) - Nous montrons à propos de la génération de code pour l'utilisation des valeurs comment les copies de valeurs de mode union se ramènent aux cas précédents.

4) - Lors de la préservation de l'accessibilité (voir ch. II.4.1, § 2.4), l'ordre séquentiel de traitement des éléments permet d'éviter les chevauchements

Nous pouvons maintenant donner un schéma pour les actions *affectation* et *copiedynamique*.

action affectation = (descr δdestination, δsource, ent typeaffectation)
co les descripteurs δdestination et δsource représentent la destination et la source de la copie ; typeaffectation est le type de la copie que l'on veut effectuer co
cas typeaffectation dans

statique : *co génération de code pour la copie de la partie statique d'une valeur. Cette génération dépend du mode de la valeur source co*

statdyn : *co génération de code pour la copie d'une valeur (parties statique et dynamique de sa représentation). La copie est effectuée élément par élément. On produit du code pour parcourir la représentation de la valeur si elle est dynamique. Pour une valeur non dynamique on produit le code de la copie avec l'action :*
affectation (δdestination, δsource, statique) co

dynamique : *co affectation (δ, fant, dynamique) co*
co génération de code pour la copie, dans l'environnement courant actif, de la partie dynamique des paramètres. La partie statique de la valeur à passer en paramètre (par identité) est en place dans le nouvel environnement.
Cette copie nécessite une allocation d'espace. On doit produire le code d'un parcours de construction de la représentation de la valeur arborescente (passage de paramètres par identité). On ne dispose que du descripteur de la destination δ décrivant la partie statique de la valeur et donnant accès à la partie dynamique dans l'ancien environnement (partie à partir de laquelle doit s'effectuer l'identité). On procède comme suit :
1) On crée temporairement dans la zone d'évaluation la nouvelle partie statique et l'on engendre le code pour l'initialiser à partir de l'ancienne. Elle est représentée par un descripteur δ'.

2) On peut alors produire le code pour construire la nouvelle valeur (δ') à partir de l'ancienne (δ) (affectation(δ', δ, identité))

3) On produit le code pour transmettre la partie statique du paramètre :

(affectation (δ, δ', statique))

co

identité : *co affectation (δ, δinitialisation, identité) co*
co génération de code pour construire une valeur par identité à partir d'une autre. La partie statique de la nouvelle valeur, allouée mais non initialisée, est représentée par δ. La valeur à partir de laquelle est faite l'identité est représentée par δinitialisation.
Cette copie (construction de la représentation d'une nouvelle valeur) nécessite une allocation d'espace pour la partie dynamique (éventuelle).
On produit le code pour un parcours de construction (mise en place des parties statiques après chaque allocation) de la valeur si elle est dynamique.
Si elle n'est pas dynamique on produit le code de copie avec l'action :

affectation (δdestination, δsource, statique)

co

fincas

finaction

action copiedynamique = (*descr* δval)

co *génération du code pour préserver l'accessibilité à une valeur lors d'un changement d'environnement à l'exécution. La partie statique de la valeur est dans le nouvel environnement (représentée par δval). Sa partie dynamique peut ne pas être accessible dans ce nouvel environnement* *co*

co * *génération du test permettant de déterminer si la partie dynamique est accessible ou non (à l'aide du pointeur sur la tête de la pile que l'on trouve dans les données de liaison de chaque environnement)*
* *génération du branchement à la fin du code de la copie, si la partie dynamique est accessible* *co*

co *génération de code pour construire un nouvel exemplaire (accessible) de la partie dynamique :*

affectation (δval, fant, dynamique)

co

co *génération de l'étiquette de fin de copie* *co*

finaction

4.3 - Actions de génération de code pour les valeurs-résultat -
(valeurs retournées par une routine, une instruction conditionnelle, etc...).

Deux actions permettent de décrire la génération de code nécessaire à la transmission des valeurs-résultat prévue dans le système à l'exécution (voir ch. II.4.1, § 2.4.3).

4.3.1 - L'action *setres* permet de produire le code de recopie, dans la zone d'évaluation, de la partie statique d'une valeur transmise comme valeur-résultat (par exemple dans le cas d'un appel de procédure) pour la rendre accessible dans le nouvel environnement.

La fonction *faireres*(δ) transforme le descripteur δ contenant les caractéristiques de mode d'une valeur pour qu'il représente cette valeur comme valeur-résultat.

```
action setres = (descr δ, δres)
        co la valeur est une valeur-résultat
            δ : descripteur décrivant les caractéristiques du mode de la
            valeur. Transférer la (partie statique de la) valeur-résultat
            dans la zone d'évaluation.
            δres : descripteur pour le résultat dans la pile statique
                   d'évaluation co
    δ ← faireres(δ)
        co faire représenter par δ une valeur en ressource-résultat co
    δres ← allocpse(taille(modedescr(δ)))
        co allocation dans la zone d'évaluation co
    δres ← complétermode(δres, modedescr(δ))
        co δres va représenter la valeur-résultat transférée dans la zone
           d'évaluation co
    affectation(δres, δ, statique)
        co transfert de la valeur co
finaction
```

4.3.2 - L'action *construirerésultat* produit le code permettant de transmettre une valeur comme valeur-résultat dans les instructions choix et les acheveurs (*exit*) bien qu'il n'y ait pas changement d'environnement. Si l'on considère le graphe de programme associé à une construction de choix (choix-entier, choix-union, si-alors-sinon) on constate que plusieurs chemins, correspondant à plusieurs états de la zone statique, sont possibles pour l'élaboration d'une telle proposition. Il est nécessaire d'obtenir le même état de la pile pour accéder, entre autres, à la valeur délivrée par la construction, quel que soit le chemin ou l'alternance élaborée.

Une solution consiste à transmettre la valeur de la série formant chaque alternance comme valeur-résultat. La génération s'effectue en deux étapes :

- au niveau de chaque alternance on produit le code de transformation de la valeur de la série en valeur-résultat,
- à la fin de la construction on produit le code pour ranger ce résultat dans la zone d'évaluation.

Cette dernière génération et la construction du descripteur synthétisé à partir du noeud font l'objet de l'action *construirerésultat*.

Remarque : En pratique, nous considérons les ressources du type registre, accumulateur, etc... comme faisant partie de la zone d'évaluation. Pour des raisons d'efficacité, la transmission des valeurs-résultat se fait en utilisant ces ressources, ce qui évite, dans la majorité des cas, le code de rangement (du résultat) dans la zone d'évaluation.

```
action construirerésultat = (descr δsérie, δres)
        co
          allouer un descripteur et lui faire représenter une valeur-
          résultat ayant les caractéristiques de celle représentée par
          δsérie. Transférer cette valeur dans la zone d'évaluation en
          modifiant δres en conséquence
        co
    descr δ
    δ ← complétermode(δ, modedescr(δsérie))
    setres(δ, δres)
finaction
```

5 - MODELES UTILISES DANS LA DESCRIPTION -

Nous donnons les modèles utilisés dans toute la description. D'autres modèles sont associés plus particulièrement à un noeud ou à un groupe de noeuds. Nous les énumérons en les regroupant.

5.1 - Modèles permettant la construction de texte de routine -

routineentête et routinefin

5.2 - Modèle appelant un texte de routine -

activerroutine

5.3 - Modèle transformant une valeur en valeur résultat -

résultat

5.4 - Modèles permettant l'utilisation d'une routine de bibliothèque -

chargement de la valeur d'un paramètre : chargeparamètre
récupération de la valeur d'un paramètre : copierésult

5.5 - Modèle d'allocation et d'initialisation d'une ressource servant de base -

allocbase

5.6 - Modèles divers -

copiant des valeurs continues : copiercontigu
sauvegardant l'accès à une zone : copieaccès

modèle routineentête = (étiquette étiq)
co code de l'en-tête d'une routine co
(ptentrée, étiq, routine,)
co définition du point d'entrée de la routine co
co à ce point est rajouté le code réalisant le changement d'environnement, c'est-à-dire l'allocation de l'espace correspondant et la mise en place des données de liaison de ce nouvel environnement co
co génération de la recopie du vecteur d'accès à partir de celui de l'environnement d'appel et du niveau statique de l'environnement. Ce niveau est fourni par une variable de l'environnement décrit par Δenvcour co
finmodèle

modèle routinefin =
co code de fin d'une routine co
(finroutine, taillepsde Δenvcour, ,)
co la taille de la zone statique de la routine est récupérée à cet endroit pour être passée en paramètre à la fonction de la bibliothèque réalisant le changement d'environnement co
finmodèle

```
modèle activerroutine = (descr δroutine)
        co appel d'une routine
           δroutine : descripteur représentant la routine à appeler
        co
    (activer, δroutine, , )
finmodèle

modèle résultat = (descr δ)
        co code pour transformer une valeur en valeur-résultat.
           L'objet possédant cette valeur est représenté par le descripteur δ
        co
    (misenres, δ, , )
finmodèle

modèle chargeparamètre = (descr δparam, δvaleur)
        co passage d'un paramètre à une routine de bibliothèque
           δparam : descripteur d'une ressource servant au passage de
                    paramètre à une routine de bibliothèque
           δvaleur : descripteur de l'objet possédant la valeur à passer
                     en paramètre
        co
    (:=, δparam, δvaleur, )
finmodèle

modèle copierésult = (descr δd)
        co récupération du résultat d'une routine de bibliothèque
           δd : descripteur de l'emplacement où le résultat doit être rangé
           δrésultbibliothèque : descripteur de la ressource contenant le
                                 résultat des routines de bibliothèque
        co
    (:=, δd, δrésultbibliothèque, )
finmodèle

modèle allocbase = (descr δ, δinit) =
        co allocation et initialisation d'une base
           δ : descripteur pour représenter la base une fois allouée et
               initialisée
           δinit : descripteur de l'initialisation de la base
        co
    (allouer, base, δinit, δ)
        co δ représente la ressource servant de base co
finmodèle
```

```
modèle copiercontigu = (descr δdestination, δsource)
        co copie de valeurs de représentation continue
           δdestination : descripteur représentant la destination
           δsource : descripteur représentant l'objet possédant la valeur
                     à transmettre
        co
    (copiecont, δdestination, δsource , )
finmodèle

modèle copieaccès = (descr δsauveaccès, δzone)
        co sauvegarde de l'accès à une zone
           δsauveaccès : descripteur de l'emplacement de rangement
           δzone : descripteur de la zone dont on veut sauvegarder l'accès
        co
    (copie@, δsauveaccès, δzone, )
finmodèle
```

6 - LES QUADRUPLETS -

Les quadruplets, nous l'avons vu, permettent de décrire les instructions ou groupes d'instructions-machine produites après transformation par un traducteur pendant la génération de code.

Nous précisons ici la liste des quadruplets utilisés pour toutes les constructions du langage, sauf celles qui concernent les opérations sur des valeurs simples (affectation, opérateurs arithmétiques, logiques, etc...) et les parcours des valeurs dynamiques.

Leur introduction ne pose aucun problème particulier il suffit d'ajouter à cette liste des quadruplets dont le code opération est l'opérateur que l'on veut coder. Le parcours des valeurs dynamiques, dépendant de la représentation utilisée, n'est décrit que dans son principe. Rappelons que les quadruplets comportent quatre champs dont le premier indique le code de l'opération à coder, et les trois autres les opérandes (qui sont en général sous forme de descripteurs). Lorsqu'un opérande n'est pas significatif il est remplacé dans la description du quadruplet par fant (fantome).

6.1 - Quadruplets associés à la génération d'instructions de contrôle dans le code objet -

6.1.1 - Définition d'étiquettes -

Nous avons supposé dans notre description que la résolution des problèmes d'adressage dûs aux étiquettes a lieu au chargement. Nous sommes amenés à engendrer des étiquettes de diverse nature (étiquettes programmes, étiquettes anonymes, points d'entrée) et des tables d'étiquettes pour les aiguillages.

quadruplet (deftabentrée, étiquette étiquette, ent nbélém, fant) =
    co définition de la table des entrées des alternances du cas
        étiquette : étiquette en symbolique de la première alternance
        nbélém : nombre d'alternances
    co
finquadruplet

quadruplet (ptentrée, étiquette étiq, ent type, fant) =
    co définition d'un point d'entrée
        étiq : étiquette (en symbolique) du point d'entrée
        type : type du point d'entrée, c'est-à-dire :
                routine, cas,...
        Il y a insertion de code pour un traitement spécifique au type du point d'entrée
    co
finquadruplet

quadruplet (défétiq, étiquette étiq, fant, fant) =
    co définition d'étiquette
        étiq : étiquette à définir (en symbolique)
    co
finquadruplet

6.1.2 - Comparaison -

quadruplet (comparer, ent type, descr δg, descr δd) =
co comparaison
δg : descripteur de l'opérande gauche
δd : descripteur de l'opérande droit
type : type de la comparaison
comparaison booléenne,
comparaison arithmétique, ...
Cette comparaison positionne une ressource-machine spécialement prévue à cet effet co
finquadruplet

6.1.3 - Branchements -

quadruplet (branchement, ent condition, étiquette étiq, fant) =
co branchement simple
condition : condition du branchement, c'est-à-dire:
faux, <, =, >, ≤, nop, inconditionnel, ≠, ...
étiq : étiquette à laquelle il faut se brancher
Cette instruction utilise la valeur de la ressource-machine positionnée par les instructions de comparaison co
finquadruplet

quadruplet (sautprog, descr δbr, fant, fant) =
co génération d'un branchement à une étiquette du programme
δbr : descripteur de l'étiquette
produit le code pour restaurer la base de l'environnement et la base du code associées à la routine contenant la définition de l'étiquette, puis produit le branchement à l'étiquette
co
finquadruplet

6.2 - Quadruplets associés à l'allocation de ressource et à quelques types d'opérations sur les valeurs entières -

Les quadruplets décrits ici donnent un exemple de ceux qu'il faut écrire pour représenter tous les opérateurs.

Nous avons utilisé, pour noter les opérations, des symboles semblables à ceux des opérateurs standards d'Algol 68. Leur traduction en séquence d'instructions machine est simple. En effet, ces séquences utilisent en général un ou deux registres de travail temporaires, qu'il est aisé d'introduire au moment de la traduction.

```
quadruplet (allouer, ent type, descr δs, descr δd) =
    co allocation et initialisation d'une ressource
       type : type de la ressource demandée (base, quelconque)
       δs : descripteur de l'objet possédant la valeur avec laquelle il
            faut initialiser la ressource
       δd : descripteur à initialiser pour qu'il représente la ressource
            allouée
       Le type de la ressource allouée précise si cette ressource (registre,
       accumulateur, ...) peut ou non servir de base pour accéder à une
       zone
    co
finquadruplet
```

```
quadruplet (:=, descr destination, descr δsource, fant) =
    co affectation d'une valeur entière
       δdestination : descripteur de la destination
       δsource : descripteur de l'objet possédant la valeur à affecter
    co
finquadruplet
```

```
quadruplet (+ :=, descr δg, descr δd, fant) =
    co addition entière et affectation à l'opérande gauche
       δg : descripteur représentant l'opérande gauche
       δd : descripteur représentant l'opérande droit
        génération du code additionnant les valeurs (de type entier) des
        deux opérandes et rangeant la valeur dans l'emplacement représenté
        par δg
    co
finquadruplet

quadruplet (- :=, descr δg, descr δd, fant) =
    co soustraction entière et affectation à l'opérande gauche co
       δg : descripteur représentant l'opérande gauche
       δd : descripteur représentant l'opérande droit
        génération de code soustrayant les valeurs (de type entier) des deux
        opérandes et rangeant la valeur dans l'emplacement représenté par δg
    co
finquadruplet

quadruplet (*+ :=, descr δgm, descr δdmga, descr δdadest) =
    co multiplication, addition et affectation de valeurs entières
       δgm : descripteur de l'opérande gauche de la multiplication
       δdmga : descripteur de l'opérande droit de la multiplication
               le résultat de l'opération est l'opérande gauche de l'addition
       δdadest : descripteur de l'opérande droit de l'addition
               le résultat de l'addition est rangé dans l'emplacement
               représenté par δdadest
    co
finquadruplet
```

Remarque : En s'inspirant des quadruplets qu'il vient de voir, le lecteur complètera aisément la liste des quadruplets nécessaires pour tous les opérateurs standard définis dans le langage.

6.3 - Quadruplets associés à la génération de code pour les textes de routine, les appels, et les valeurs résultat -

Le quadruplet *finroutine* permet d'ajouter, dans le code de fin de routine, la taille de la zone statique utilisée par la routine. Au moment du chargement cette information est transformée en une séquence d'instructions, insérées en tête du code de la routine, allouant l'espace pour cette zone dans la pile. Cette solution permet de prendre en compte, sans trop d'artifices, les informations que l'on ne connaît qu'en fin de la génération du code correspondant à un texte de routine ; elle utilise également la phase de chargement que l'on ne peut guère éviter. Dans le cas d'une génération du code directement dans la mémoire suivie d'une exécution, (compilateur "load and go") le problème ne se pose plus car on dispose à chaque instant de tout le code.

```
quadruplet (finroutine, ent taille, fant, fant) =
    co fin d'une routine
       taille : taille totale de la zone statique. Cette taille est transmise
                pour pouvoir, à l'exécution, réserver, en début de routine,
                la place pour la zone statique
    co
finquadruplet
```

```
quadruplet (activer, descr &rout, fant, fant) =
    co appel d'une routine
       &rout : descripteur de la routine à appeler
    co
finquadruplet
```

```
quadruplet (appelbiblio, ent type, fant, fant) =
    co appel d'une routine de bibliothèque
       type : numéro de la routine de bibliothèque à appeler
              allocationlocale, allocationglobale, etc...
    co
finquadruplet
```

```
quadruplet (misenres, descr δ, fant, fant) =
    co transformation d'une valeur en valeur-résultat
       δ : descripteur représentant la valeur à transformer co
finquadruplet

quadruplet' (copiecont, descr δd, descr δs, fant) =
    co copie d'une zone continue
       δd : descripteur de la destination
       δs : descripteur de la zone à copier
       La longueur sur laquelle la copie se réalise fait partie des
       informations contenues dans les descripteurs
    co
finquadruplet

quadruplet (exécuter, ensemble quadruplet quad, descr δchoix, fant) =
    co exécution contrôlée d'un des quadruplets de quad
       quad : ensemble de quadruplets
       δchoix : descripteur de l'objet contrôlant l'exécution
       Suivant la valeur possédée par cet objet il y a exécution de(s)
       l'instruction(s)-machine correspondant à l'un des quadruplets de
       l'ensemble
    co
finquadruplet

quadruplet (option, ent type, fant, fant) =
    co début de chargement conditionnel
       type : caractéristique des vérifications effectuées par le code
              qui suit
    co
finquadruplet

quadruplet (finoption, fant, fant fant) =
    co fin de chargement conditionnel
    co
finquadruplet
```

Nous avons décidé de produire le code pour effectuer toutes les vérifications ; ce code n'est cependant chargé que sur option. Ceci évite des compilations, le code produit est toutefois plus volumineux. A titre d'exemple de vérifications citons :

- les indices compris entre les bornes d'un tableau,
- les pointeurs non initialisés,
- les tableaux fantomes,
- etc....

6.4 - Quadruplet de copie d'accès -

quadruplet (copie@, descr δd, descr δs, fant) =
co rangement de l'accès à une zone
δd : descripteur de l'emplacement de rangement
δs : descripteur de la zone dont il faut ranger l'accès
co
finquadruplet

7 - CONCLUSION -

L'explication détaillée que nous venons de faire concernant toutes ces notations et conventions nous permet, dans les chapitres suivants, de donner noeud par noeud la description de la phase de génération de code.

Les notions que nous avons introduites ne trouvent leur pleine justification qu'au cours de cette description. C'est pourquoi nous conseillons au lecteur de se reporter fréquemment à ce chapitre pendant la lecture des chapitres suivants.

II.4.6 - DESCRIPTION -

La génération de code est réalisée au cours de la "réduction" de l'arbre abstrait. Dans la description qui suit, la fonction de génération (notée *élaborer*) est décrite récursivement et utilise une fonction de parcours de l'arbre abstrait.

Elle n'est pas explicitée globalement mais définie pour chaque noeud et utilisée dans le traitement des noeuds ayant des opérandes.

Pour la décrire nous avons regroupé les noeuds de l'arbre abstrait en sept classes :

- structures de contrôle,
- blocs et routines,
- activations,
- déclarations et générateurs,
- constructions de valeurs,
- notations et valeurs par défaut,
- autres constructions.

1 - TRAITEMENT DES STRUCTURES DE CONTROLE DE L'EXECUTION -

Nous avons séparé les structures de contrôle délivrant, en Algol 68, un résultat "sémantiquement intéressant" pour le programmeur, de celles qui n'en fournissent pas.

1.1 - Traitement des instructions *choix (booléen, entier, union)* et des *acheveurs (exit)*.

Pendant l'exécution, ces structures offrent plusieurs alternances. Une seule de ces alternances est prise et la valeur retournée par cette structure est celle fournie par l'alternance choisie.

Pour ces instructions il est nécessaire de transformer la valeur retournée par chaque alternance possible, en valeur-résultat pour qu'elle puisse être utilisée par la suite du programme. En effet, même si les caractéristiques de chaque valeur possible sont les mêmes, les fonctions d'accès à leur représentation sont généralement différentes (dans le cas contraire, la possibilité d'avoir plusieurs alternances perd de son intérêt) et ont donc besoin d'être "homogénéisées".

1.1.1 - Instruction choix-booléen -

(choix-booléen)

Plus communément connue sous le nom d'instruction conditionnelle, cette instruction s'écrit :

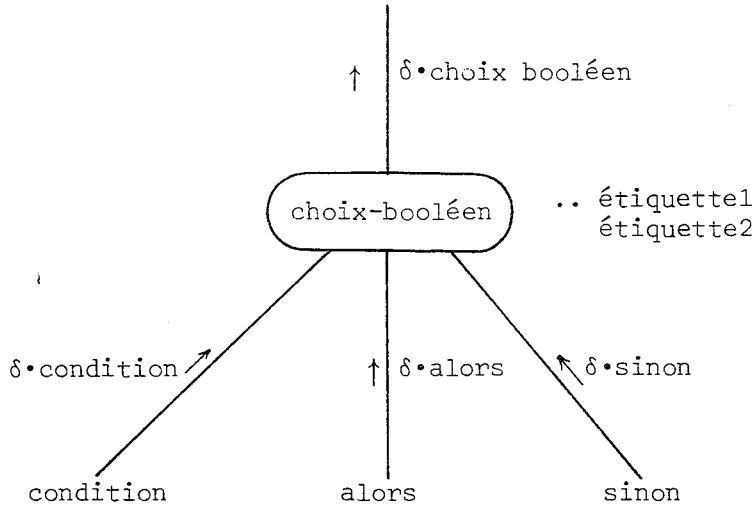
si ... *alors* ... *sinon* ... *fsi*

La génération du code correspondant va nécessiter l'emploi d'étiquettes

- une pour étiqueter la partie *sinon*
- une pour étiqueter ce qui suit l'instruction conditionnelle

Le traitement consiste à engendrer

- le calcul de la valeur booléenne
- le test de cette valeur et le branchement correspondant à l'alternance *sinon*
- le code de l'alternance *alors* suivi d'un branchement inconditionnel à l'étiquette de fin
- l'étiquette de l'alternance *sinon* et le code de cette alternance
- l'étiquette de fin



décorations

étiquette1 : étiquette réservée pour la partie *sinon* de l'instruction *choix-booléen*

étiquette2 : étiquette réservée pour l'instruction suivant l'instruction *choix-booléen*

séquence

descr δ

sauvegarde

co sauvegarde des ressources co

élaborer (condition)

bchoix$_1$ *(δ•condition)*

co génération du test du booléen et du branchement à la partie sinon co

élaborer (alors)

bchoix$_2$ *(δ•alors)*

co génération du branchement à la fin et de l'étiquette de la partie sinon co

élaborer (sinon)

bchoix$_3$ *(δ•sinon)*

co génération de l'étiquette de fin co

construirerésultat (δ•sinon, δ)

co construction d'une valeur-résultat co

δ•choix-booléen ← δ

finséquence

Remarque :
L'instruction *si ... alors ... fsi* est équivalente à l'instruction *si ... alors ... sinon fant fsi*. Le noeud (choix-booléen) a donc toujours trois opérandes non vides.

modèle bchoix1 = (descr δcond)
co δcond est le descripteur représentant la valeur de la partie booléenne de l'instruction conditionnelle co
(comparer, bool, δvrai, δcond)
co comparaison avec la valeur "vrai" co
(branchement, faux, étiquette1 de (choix-booléen) ,)
co branchement au code associé à la partie sinon co
finmodèle

modèle bchoix2 = (descr δalors)
co δalors est le descripteur représentant la valeur de la partie alors de l'instruction conditionnelle co
(misenres, δalors, ,)
(branchement, inconditionnel, étiquette2 de (choix-booléen) ,)
co branchement à la fin du code de l'instruction conditionnelle co
(defetiq, étiquette1 de (choix-booléen) , ,)
co étiquette du code associée à la partie sinon co
finmodèle

modèle bchoix3 = (δsinon)
co δsinon est le descripteur représentant la valeur de la partie sinon de l'instruction conditionnelle co
(misenres, δsinon, ,)
(defetiq, étiquette2 de (choix-booléen) , ,)
co étiquette de fin du code de l'instruction conditionnelle co
finmodèle

1.1.2 - Instructions choix entier et choix union -

Ces deux instructions ont une syntaxe très voisine. Elles s'écrivent :

cas ... dans ..., ..., ..., ... *sinon ... fcas*

liste d'unités séparées par des virgules.

Pour les noeuds correspondants nous avons donc un opérande ayant une structure de liste.

Chaque élément de la liste correspond à une unité de la liste formant la partie *dans* de l'instruction,et est représenté par un noeud (élément entier) ou (élément-de-conformité) (suivant qu'il s'agit du *choix-entier* ou du *choix-union*) dont l'opérande est l'arbre de l'unité correspondante.

(choix-entier)

Etant donné qu'il y a plusieurs alternances, le code à engendrer nécessite des étiquettes.

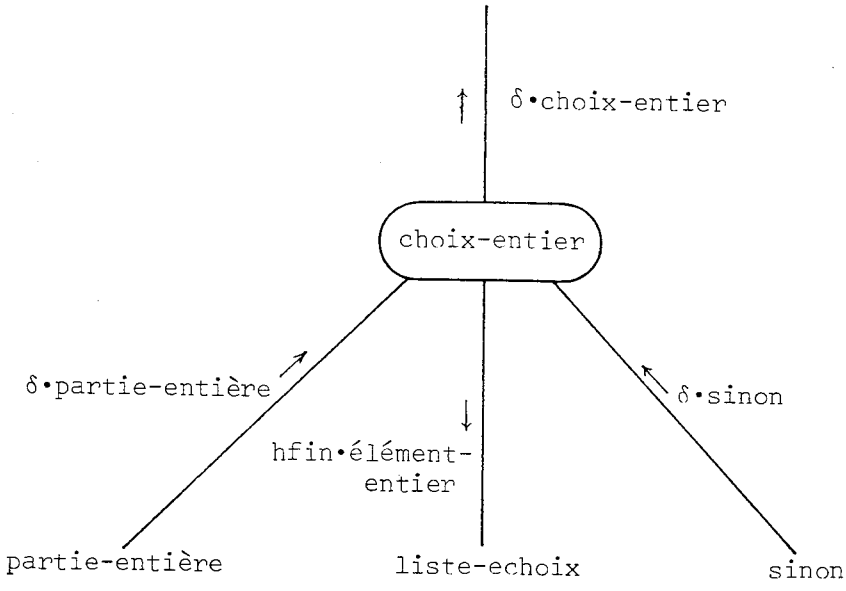
Le choix de l'alternance étant fait suivant la valeur entière délivrée par la partie *cas*, nous construisons dans le code une table contenant les étiquettes de chaque alternance de la partie *dans*. A l'exécution (une fois testée la valeur entière,pour décider s'il faut ou non choisir l'alternance *sinon*), en indexant cette table avec la valeur entière, on obtient le point d'entrée de l'alternance à exécuter.

Le traitement consiste donc à produire :

- le calcul de la valeur entière,
- le test de cette valeur (choix de l'alternance *sinon*),
- la table des alternances précédée de son indexation,
- le code de chaque alternance, en l'encadrant chaque fois de la définition de l'étiquette correspondante et du branchement à l'étiquette de fin. Chaque valeur fournie est transformée en valeur-résultat,
- l'étiquette de l'alternance *sinon* suivie du code de l'alternance,
- l'étiquette de fin.

Les étiquettes utilisées sont :

- l'étiquette de la table des alternances,
- l'étiquette de fin,
- l'étiquette de l'alternance *sinon*,
- l'étiquette de chaque alternance de la partie *dans*. Cette étiquette est une décoration de chaque noeud (élément-entier) .



décorations

étiquette1 = étiquette réservée pour la table des alternances
étiquette2 = étiquette réservée pour l'instruction suivant l'instruction *choix-entier*
étiquette3 = étiquette de l'alternance *sinon*
étiquette4 = étiquette réservée pour la première alternance et servant au remplissage de la table des alternances
nbéléments = nombre d'éléments constituant la liste liste-echoix

informations à hériter

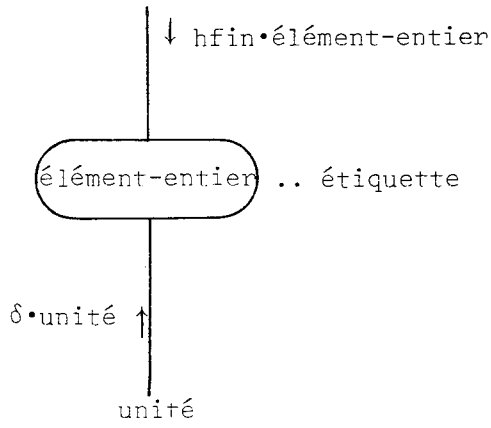
hfin.élément-entier représentant l'étiquette de fin (étiquette2) est héritée sur tous les éléments de la liste liste-echoix.

```
séquence
    descr δconstante, δ
        co descripteur pour le nombre d'alternances co
    sauvegarde
        co sauvegarde des ressources co
    δconstante←constante(nbéléments de (choix-entier) )
    élaborer (partie-entière)
    echoix1 (δ•partie-entière, δconstante)
        co génération du branchement et de la table des entrées d'alter-
           nances co
    pour tout arbre élément-entier ∈ liste-echoix
    faire
        séquence
            hfin•élément-entier ← étiquette2 de (choix-entier)
            élaborer (élément-entier)
                co traitement de l'alternance co
        finséquence
    finfaire
        echoix2
            co génération de l'étiquette de sinon co
        élaborer (sinon)
        résultat (δ•sinon)
            co transformation de la valeur retournée par sinon en valeur-
               résultat co
        echoix3
        construirerésultat (δ•sinon, δ)
        δ•choix-entier ← δ
finséquence
```

Remarque :

L'instruction *cas ... dans alternance$_1$, ..., alternance$_n$ fcas* est équivalente à l'instruction *cas ... dans alternance$_1$, ..., alternance$_n$ sinon fant fcas*

```
modèle echoix1 = (descr δvarcas, δconstante)
        co δvarcas : descripteur représentant la variable de contrôle du cas
           δconstante : descripteur représentant le nombre d'alternances co
    (comparer, arith, δvarcas, δzéro)
    (branchement, ≤, étiquette3 de (choix-entier) , )
    (comparer, arith, varcas, constante)
    (branchement, >, étiquette3 de (choix-entier) , )
        co branchements conditionnels à la partie sinon co
    (ptentrée, étiquette1 de (choix-entier), casentier, )
        co insertion du code permettant, à partir du numéro d'alternance,
           d'effectuer un branchement à la bonne alternance. Définition de
           l'étiquette associée à la table des entrées d'alternances co
    (deftabentrée, étiquette4 de (choix-entier) , nbéléments de (choix-entier), )
        co définition de la table des entrées de chaque alternance
           l'étiquette transmise est celle de la première alternance co
finmodèle

modèle echoix2 =
        co définition de l'étiquette d'entrée partie sinon co
    (defetiq, étiquette3 de (choix-entier), , )
finmodèle

modèle echoix3 =
        co définition étiquette de fin co
    (defetiq, étiquette2 de (choix-entier) )
finmodèle
```



décoration

étiquette = étiquette réservée pour l'alternance.

information héritée

hfin•élément-entier représente l'étiquette de fin de l'instruction *choix-entier*.

séquence

echoixliste1

co génération de l'étiquette d'alternance co

élaborer(unité)

co synthétisation de δ.unité co

echoixliste2 (hfin•élément-entier, δunité)

co génération pour transformer la valeur en valeur-résultat et génération du branchement à la fin co

finséquence

modèle echoixliste1 =

co génération de l'étiquette d'entrée de l'alternance co

(defetiq, étiquette de (élément-entier) , ,)

finmodèle

modèle echoixliste2 = (étiquette hfincasetiq, descr δalt)

co hfincasetiq : numéro d'étiquette définissant la fin du cas

δalt : descripteur de la valeur à transformer en valeur-résultat cc

(misenres, δalt, ,)

(branchement, inconditionnel, hfincasetiq,)

finmodèle

choix-union

Etant donné qu'il y a plusieurs alternances, le code à produire nécessite l'emploi d'étiquettes.

Le choix de l'alternance étant fait suivant le mode courant (à l'exécution) de la valeur unie délivrée par la partie *cas*, nous construisons dans le code une table contenant les étiquettes de chaque alternance de la partie *dans*. Il n'existe pas forcément une alternance spécifique de chaque mode que peut prendre la valeur unie. A plusieurs modes peut correspondre une seule alternance.

Exemple :

```
début
    union (ent, rep car, bool, proc(ent) neutre, reel, compl) uni = ...
      :
    cas uni
    dans
        (ent i) : ... ,
        (union (réel, compl) urc) : ... ,
        (rep car) : ...
    sinon
        ...
    fcas ;
      :
fin
```

L'alternance *sinon* est prise lorsque le mode de *uni* est *bool* ou *proc(ent) neutre*. De même la deuxième alternance est choisie lorsque le mode de *uni* est *réel* ou *compl*. Remarquons qu'il est possible d'adjoindre un identificateur au déclareur d'une alternance. Une identité est alors opérée entre la valeur unie et la déclaration d'identificateur ainsi constituée.

Le principe appliqué à l'instruction *choix-entier* n'est plus suffisant car il n'y a pas de correspondance biunivoque entre l'ensemble des modes possibles de la valeur unie et l'ensemble des alternances.

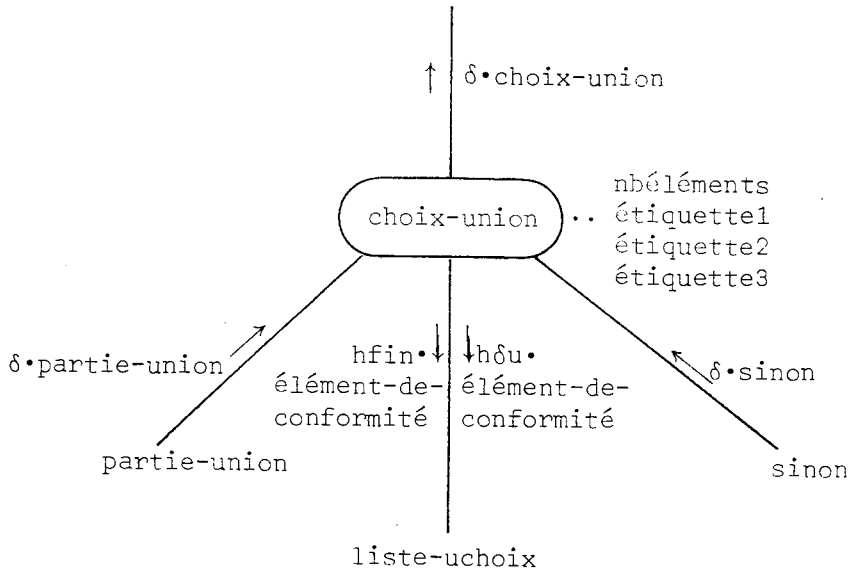
C'est pourquoi, en plus de la table des alternances, nous construisons une deuxième table (table des sous-modes) fournissant pour chaque mode possible le numéro de l'alternance correspondante. Pendant l'exécution, en indexant cette table avec le rang du mode courant de l'union (ce qui implique d'ordonner les sous-modes composant un mode *union*), nous obtenons l'index dans la table des alternances. Il ne reste plus qu'à indexer cette table pour connaître le point d'entrée de l'alternance à exécuter.

Le traitement consiste donc à produire :

- le calcul de la valeur unie,
- la table des alternances précédée des indexations nécessaires à partir du rang du mode (la table des sous-modes est une constante de compilation et n'est pas mise dans le code produit mais dans une table des constantes),
- le code de chaque alternance de la partie *dans* en l'encadrant à chaque fois de la définition de l'étiquette correspondante et du branchement à l'étiquette de fin. Chaque valeur fournie est transformée en valeur-résultat,
- l'étiquette de l'alternance *sinon* suivie du code de l'alternance,
- l'étiquette de fin.

Les étiquettes utilisées sont :

- l'étiquette de la table des alternances,
- l'étiquette de fin,
- l'étiquette de l'alternance *sinon*,
- l'étiquette de chaque alternance de la partie *dans*. Cette étiquette est une décoration de chaque noeud (élément-de-conformité) .



décorations

étiquette1 = étiquette réservée pour la table des alternances.

étiquette2 = étiquette réservée pour l'instruction suivant l'instruction *choix-union*.

étiquette3 = étiquette de l'alternance *sinon*. Elle joue aussi le rôle de première étiquette et va servir au remplissage de la table des alternances.

nbéléments = nombre d'alternances constituant la liste liste-uchoix.

informations à hériter

Les descripteurs de valeur hfin•élément-de-conformité représentant l'étiquette de fin (étiquette2) et hδu•élément-de-conformité représentant la valeur unie sont hérités sur toutes les alternances de liste-uchoix.

```
séquence
    descr δconstante1, δidunion, δ,
        co descripteur pour la table des sous-modes et pour le mode courant
           de la valeur unie co
    sauvegarde
        co sauvegarde des ressources co
    élaborer (partie-union)
    δconstante1 ← fairetabssmode(modedescr(δ•partie-union), liste-uchoix)
        co construction de la table des sous-modes et rangement dans la
           table des constantes co
    δidunion ← construireidunion(δ•partie-union)
    uchoix1(δidunion, δconstante1)
        co génération du branchement et de la table des entrées d'alternances

    pour tout arbre élément-de-conformité ∈ liste-uchoix
    faire
        séquence
        hfin•élément-de-conformité←étiquette2 de (choix-union)
        hδu•élément-de-conformité←δ•partie-union
            co transmission de l'étiquette de fin et du descripteur de la
               valeur unie co
        élaborer (élément-de-conformité)
            co génération de code pour l'alternance co
        finséquence
    finfaire
    uchoix2
        co définition de l'étiquette de l'alternance sinon co
    élaborer(sinon)
    résultat(δ•sinon)
        co transformation de la valeur retournée en valeur-résultat co
    uchoix3
        co définition de l'étiquette de fin co
    construirerésultat(δ•sinon, δ)
    δ•choix-union←δ
finséquence
```

Remarque :

L'instruction *cas ... dans alternance$_1$, ..., alternance$_n$ fcas* est équivalente à l'instruction :

cas ... dans alternance$_1$, ..., alternance$_n$ sinon fant fcas

fonction fairetabssmode = (type modeuni, arbre listealternance) descr

co - Construit la table donnant pour chaque sous-mode de modeuni le numéro de l'alternance. Ce numéro sert d'index dans la table des alternances pour obtenir le point d'entrée correspondant au mode courant (à l'exécution) de la valeur unie.

- Range cette table dans la table des constantes.

- Alloue et initialise un descripteur décrivant cette table.

- Retourne ce descripteur comme résultat.

co

finfonction

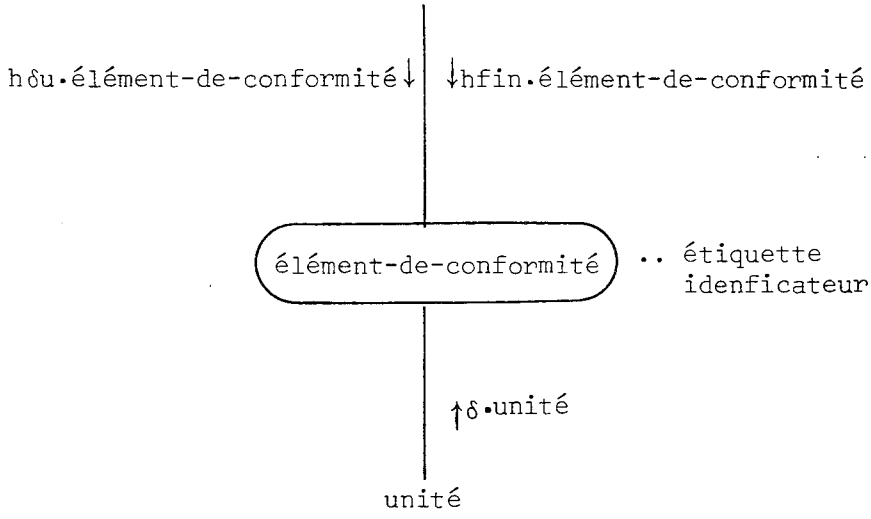
fonction construireidunion = (descr δunion) descr

co - Construit, à partir du descripteur de la valeur unie, le descripteur représentant l'en-tête de cette valeur. Cet en-tête contient une identification du mode courant.

co

finfonction

```
modèle uchoix1 = (descr δunion, δconstante)
        co
          δunion : descripteur représentant le mode courant de la valeur
          δconstante : descripteur représentant la table des sous-modes
        co
    co
      génération du code indexant la table des sous-modes
      Ce code récupère le numéro de l'alternance correspondante et ramène
      au cas d'une instruction choix entier
    co
    (ptentrée, étiquette1 de (choix-union) , casunion, )
        co insertion du code permettant, connaissant le numéro d'alternance,
           d'effectuer un branchement au code correspondant. Définition de
           l'étiquette associée à la table des entrées d'alternances co
    (deftabentrée, étiquette3 de choix-union, nbéléments de (choix-union) + 1,

        co génération de la table des alternances co
finmodèle

modèle uchoix2 =
        co définition de l'étiquette de l'alternance sinon co
    (defetiq, étiquette3 de (choix-union) , , )
finmodèle

modèle uchoix3 =
        co définition de l'étiquette de fin co
    (defetiq, étiquette2 de (choix-union) , , )
finmodèle
```



décorations

étiquette : étiquette réservée pour l'alternance
identificateur : elle correspond à l'identificateur (s'il existe) apparaissant dans la déclaration de l'alternance

informations héritées

hδu•élément-de-conformité représente le descripteur de la valeur unie
hfin•élément-de-conformité représente l'étiquette de fin

```
séquence
    descr δ
    uchoixliste1
        co génération de l'étiquette d'alternance co
    si idfprésent(identificateur de (élément-de-conformité) )
    alors
        séquence
            δ ← descridf(identificateur de (élément-de-conformité) )
            affectation(δ, hδu•élément-de-conformité)
                co génération de code réalisant l'identité avec la valeur
                   unie co
        finséquence
    finsi
    élaborer (unité)
        co synthétisation de δ•unité co
    uchoixliste2 ( hfin•élément-de-conformité, δ•unité)
        co génération pour transformer la valeur en valeur-résultat et
           génération du branchement à la fin co
finséquence

fonction idfprésent = (identificateur b) bool
        co
          teste la présence de la décoration identificateur co
finfonction

modèle uchoixliste1 =
        co définition de l'étiquette de début d'alternance co
    (defetiq, étiquette de (élément-de-conformité) , , )
finmodèle

modèle uchoixliste2 = (étiquette hfincasetiq, descr δalt)
        co hfincasetiq : étiquette de fin
           δalt : descripteur de l'unité de l'alternance co
    (misenres, δalt, , )
        co mise en valeur-résultat de la valeur de l'unité co
    (branchement, inconditionnel, hfincasetiq, )
        co génération du branchement à l'étiquette de fin co
finmodèle
```

1.1.3 - Acheveur -

exit

Cette structure de contrôle possède deux branches, la deuxième étant déjà étiquetée dans le programme.

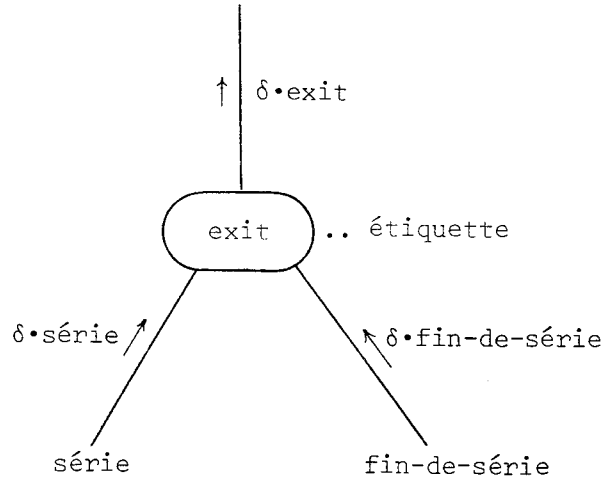
Nous avons donc besoin d'une étiquette pour "sauter par dessus" la deuxième branche.

Le traitement consiste à engendrer :

- le code pour la première branche avec mise en ressource de la valeur fournie
- le branchement à l'étiquette de fin (décoration étiquette)
- le code pour la deuxième branche avec mise en ressource de la valeur fournie
- l'étiquette de fin.

Exemple :

```
début
    ent a, b, c, d, e ; bool test ;
     .
     .
     .
    e := (... si test alors allera etiq1 fsi ;
     .          .
     .          .
                .
          a + b exit etiq1 :
                .
                .
                .
            c * d) ;
     .
     .
     .
fin
```



décoration

étiquette : étiquette de fin

séquence

descr δ
élaborer(série)
exit1(δ•série)
co génération de la mise en ressource-résultat et du branchement à la fin co
élaborer(fin-de-série)
exit2(δ•fin-de-série)
co génération de la transformation en résultat et de l'étiquette de fin co
construirerésultat(δ•fin-de-série, δ)
δ•exit ← δ

finséquence

```
modèle exit1 = (descr δsérie)
        co δsérie : descripteur de la série co
    (misenres, δsérie, , )
    (branchement, inconditionnel, étiquette de( exit ), )
        co branchement à l'étiquette de fin de série co
finmodèle
```

```
modèle exit2 = (descr  δfinsérie)
        co δfinsérie : descripteur de fin de série co
    (misenres, δfinsérie, , )
    (defetiq, étiquette de( exit ), , )
        co définition étiquette de fin de série co
finmodèle
```

1.2 - Structures de contrôle ne délivrant pas de valeurs "sémantiquement intéressantes" -

Il s'agit de l'instruction d'itération (qui ne délivre pas de valeur), et de l'instruction de branchement (dont la valeur n'est pas exploitable lorsqu'elle n'est pas neutralisée).

Contrairement aux précédentes, ces structures de contrôle n'offrent pas plusieurs alternances distinctes.

Exemple :

```
début
    réel x := 1.0 ;
    étiq1 : imprimer (x) ;
    x := allera étiq1 ;
      .
      .
      .
fin
```

Le mode (de la valeur) demandé pour la partie droite de l'affectation est *réel*. La construction *allera étiq1* est considérée comme étant du mode demandé. Pendant l'exécution, aucune valeur n'est affectée puisqu'un déroutement a lieu. En conséquence l'exécution de ce programme imprime indéfiniment la valeur *1.0*.

1.2.1 - Instruction d'itération -

(itération)

Cette instruction est plus connue sous le nom "d'instruction pour".

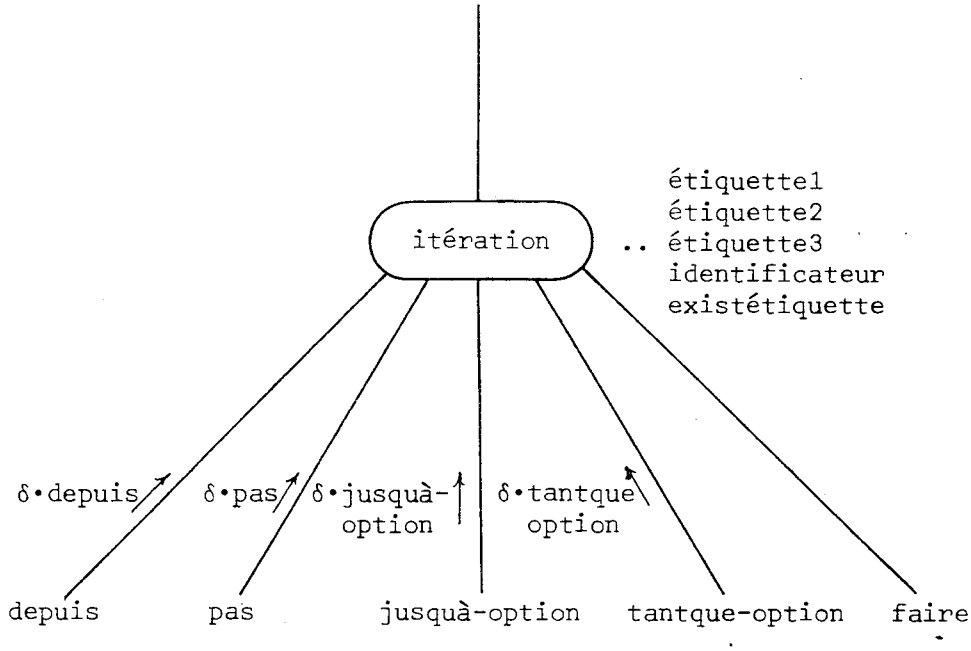
L'initialisation, le pas et la limite ne sont à calculer qu'une seule fois (ce qui n'est pas génant pour l'initialisation). Pour pouvoir réutiliser les valeurs du pas et de la limite à chaque "tour de boucle" il est nécessaire de les sauvegarder temporairement (dans la zone d'évaluation).

Une fois les allocations temporaires réalisées, le traitement consiste à produire :

(1) - le code évaluant la partie *depuis* (initialisation),
(2) - l'initialisation de la variable contrôlée avec cette valeur,
(3) - le code évaluant la partie *pas* (pas),
(4) - la sauvegarde de cette valeur,
(5) - le code évaluant la partie *jusquà* (limite),
(6) - la sauvegarde de cette valeur,
(7) - le branchement à l'étiquette du test de la variable contrôlée (contre la limite) (10) ,
(8) - l'étiquette de début de boucle,
(9) - l'incrémentation de la variable contrôlée (avec le pas),
(10) - l'étiquette du test de la variable contrôlée,
(11) - le test de la variable contrôlée contre la limite,
(12) - le branchement, en cas de supériorité, à l'étiquette de fin,
(13) - le code évaluant la partie *tant que*,
(14) - le test de la valeur booléenne fournie,
(15) - le branchement, si faux, à l'étiquette de fin,
(16) - le code pour la partie *faire*,
(17) - le branchement inconditionnel en début de boucle (8),
(18) - l'étiquette de fin.

Nous avons donc besoin de trois étiquettes :

- l'étiquette de début de boucle (8),
- l'étiquette de test de la variable contrôlée (10),
- l'étiquette de fin (18).



Remarque :

Certains fils peuvent être omis.

Pour *depuis* et *pas* la valeur par défaut (1) est insérée dans l'arbre lorsque ces unités ont été omises par le programmeur.

Pour *jusquà* et *tantque* les opérandes sont représentés par ⊗ lorsqu'ils ne sont pas précisés dans le programme.

décorations

étiquette1 = étiquette réservée pour le début de la boucle
étiquette2 : étiquette réservée pour le test de la variable contrôlée
étiquette3 : étiquette de fin
identificateur : décoration correspondant à la variable contrôlée
existétiquette : indication booléenne indiquant la présence ou l'absence d'au moins une unité étiquetée dans les séries constituant les branches tantque et faire.

```
séquence
    descr δvarcont, δpsepas, δpsejusquà
        co descripteurs de la variable contrôlée et des sauvegardes du
           pas et de la limite co
    δvarcont ← descridf(identificateur de (itération) )
        co construction du descripteur de la variable contrôlée co
    δpsepas ← allocpse(longent)
    δpsepas ← complétermode(δpsepas, modeent)
        co allocation d'espace dans la pile statique d'évaluation et
           initialisation du descripteur pour qu'il représente la sauvegarde
           du pas co
    δpsejusquà ← allocpse(longent)
    δpsejusquà ← complétermode(δpsejusquà, modeent)
        co allocation d'espace et initialisation du descripteur pour la
           sauvegarde de la limite co
        si existétiquette de (itération)
            alors sauvegarde
                co sauvegarde des ressources s'il existe une unité
                   étiquetée co
        finsi
        collatéral
            séquence
                élaborer(depuis)
                affectation(δvarcont, δ•depuis, statique)
                    co génération de l'initialisation de la variable
                       contrôlée co
            finséquence
            séquence
                élaborer(pas)
                affectation(δpsepas, δ•pas, statique)
                    co génération de la sauvegarde du pas co
            finséquence
        si nonnil(jusquà-option) alors
            séquence
                élaborer(jusquà-option)
                affectation(δpsejusquà, δ•jusquà-option, statique)
                    co génération de la sauvegarde de la valeur de
                       la limite co
            finséquence
        finsi
        fincollatéral
```

```
        iter1(δvarcont, δpsepas)
            co génération de l'incrémentation de la variable contrôlée et
               génération des étiquettes de début de boucle co
        si nonnil(jusquà-option) alors
            iter2(δvarcont, δpsejusqua, δpsepas)
                co génération de la comparaison avec la valeur de
                   unité-jusquà et de l'instruction de branchement co
        finsi
        si nonnil(tantque-option) alors
            élaborer(tantque-option)
            iter3(δ•tantque-option)
        finsi
        élaborer(faire)
        iter4
            co génération du branchement aux instructions de contrôle
               de l'itération, et définition de l'étiquette de fin
               d'itération co
finséquence

modèle iter1 = (descr δvarcontrôle, δpas)
        co δvarcontrôle : descripteur de la variable de contrôle
           δpas : descripteur du pas co
        co ce modèle est destiné à modifier le contenu de la variable
           de contrôle co
    (branchement, inconditionnel, étiquette2 de (itération) , )
    (defetiq, étiquette1 de (itération) , , )
    (+ := , δvarcontrôle, δpas, )
    (defetiq, étiquette2 de (itération) , , )
finmodèle
```

modèle iter2 = (descr δvarcontrôle, δjusquà, δpas)

co δvarcontrôle : descripteur de la variable de contrôle
δjusquà : descripteur de la borne supérieure
δpas : descripteur du pas co

co test de la variable de contrôle et poursuite ou fin de l'itération co

(comparer, arith, δvarcontrôle, δjusquà)

(exécuter, { *(branchement, >, étiquette3 de* (itération) *,)*
(branchement, <, étiquette3 de (itération) *,)*
(branchement, nop, étiquette3 de (itération) *,) }* *, δpas,)*

co ce quadruplet a pour effet de décrire l'exécution de l'une des instructions précisées dans l'ensemble du deuxième argument (noté par { ... }) suivant la valeur représentée par le troisième argument.
Ici on effectue l'un des trois branchements suivant que le pas est positif, négatif ou nul : co

co ce modèle n'est pas appelé si l'opérande unité-jusquà est absent co

finmodèle

modèle iter3 = (descr δtantque)

co δtantque est le descripteur représentant la valeur de l'unité tantque co

(comparer, bool, δvrai, δtantque)

co comparaison avec la valeur "vrai" co

(branchement, faux, étiquette3 de (itération) *)*

co branchement à la fin si faux co

finmodèle

modèle iter4 =

(branchement, inconditionnel, étiquette de (itération) *)*

co branchement en début de boucle co

(defetiq, étiquette3 de (itération) *, ,)*

co définition de l'étiquette de fin co

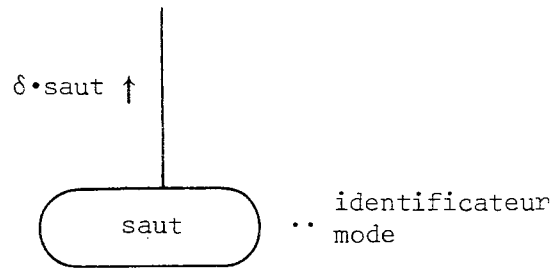
finmodèle

1.2.2 - Instruction de branchement -

(saut)

Le traitement réalisé par le code produit se décompose en deux parties :

- le chargement de la base du code et de la base de l'environnement correspondant à la définition de l'étiquette programme utilisée,
- le branchement à l'instruction étiquetée.



décorations

identificateur : décoration correspondant à l'étiquette de branchement
mode : mode demandé par le contexte pour l'instruction de branchement.

séquence

saut(descridf(identificateur de (saut)))

co construction du descripteur représentant l'étiquette de branchement co

co génération du chargement de la base du code et de la base de l'environnement. Génération du branchement co

δ•saut ← fairefantome(mode de (saut))

finséquence

Remarque :

Il y a construction d'un descripteur représentant l'instruction de branchement. Le mode demandé pour cette instruction *allera* pouvant être quelconque, le descripteur va finalement représenter une valeur *fant* du type demandé. Rappelons que, si le mode demandé est celui d'une procédure sans paramètre, un noeud (routine) a été inséré avec comme opérande le noeud (saut) .

modèle saut = (descr δbranch)

co δbranch est le descripteur représentant l'étiquette du programme à laquelle il faut se brancher co

(sautprog, δbranch, ,)

co correspond à la séquence :

- *restauration de la base de l'environnement à partir de la fonction d'accès fournie par le descripteur représentant l'étiquette,*
- *restauration de la base du code à partir des données de liaison de l'environnement dans lequel on se branche,*
- *branchement à l'étiquette programme donnée co*

finmodèle

Nous avons décrit l'ensemble du traitement des structures de contrôle à l'exécution. Ces structures sauf le *choix-union* n'apportent pas d'idées vraiment nouvelles, par rapport à Algol 60, et ne posent pas de problèmes fondamentaux aux écrivains de compilateurs.

2 - BLOCS ET ROUTINES -

C'est en principe dans les instructions bloc et routine qu'interviennent la gestion des environnements et les problèmes de transmission de valeurs entre ces environnements.

La structure de bloc n'apparaît pas que dans les noeuds (début) et (routine), mais aussi dans les noeuds (choix-booléen), (choix-union) et (itération) ; elle est attachée à certains opérandes de ces noeuds (par exemple les opérandes alors ou sinon du noeud (choix-booléen)).

Le fait d'avoir ramené la gestion des environnements au niveau des routines dans le système à l'exécution permet de considérer les instructions choix et l'instruction d'itération comme des structures de contrôle de l'exécution.

La gestion du vecteur d'accès (display), associée au traitement des routines, ne présente pas de difficultés particulières. Sa mise en place est faite, à l'exécution, dès l'entrée dans la routine, de la manière suivante :

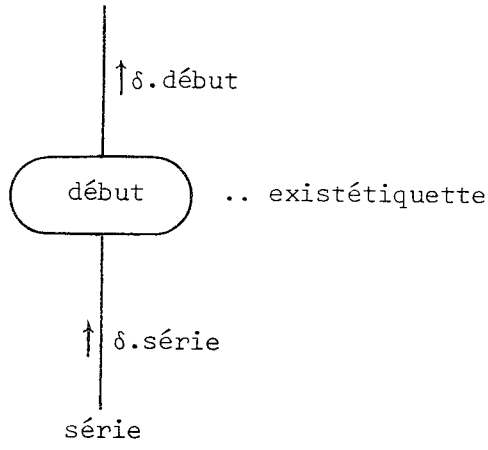
Si n est le niveau statique de la routine (ce niveau est une information appartenant à l'environnement de compilation) les actions suivantes sont effectuées :

- Recopier les n-1 premiers éléments à partir du vecteur d'accès de l'environnement d'appel
- Mettre dans le $n^{ième}$ élément l'accès à l'environnement de la routine.

Remarque :
L'utilisation d'un vecteur d'accès non local est moins coûteuse à l'entrée d'une routine mais sa restauration à la sortie est plus complexe.

2.1 - (début)

Ce noeud n'a plus,au niveau de la génération de code,qu'une signification très limitée. Le traitement consiste à engendrer le code correspondant aux unités et déclarations qui lui sont attachées.



décorations

existétiquette : cette valeur booléenne indique la présence ou l'absence d'au moins une unité étiquetée dans la série. (Les incidences sur la gestion des ressources d'une telle situation ont été étudiées au paragraphe 2.2.1 du chapitre II.4.4).

Exemple :

```
début
   ent a ;
   l: a:=1 ; ..... ; allera l ; .....
fin
```

Remarque :

Les constituants de la série sont:toutes les déclarations et instructions apparaissant dans le bloc. En particulier,toutes les déclarations nécessitent une élaboration en Algol 68,et non seulement celles qui comportent une partie formée d'instructions comme les déclarations de tableaux (exemple: *[1: (ent i ; lire (i) ; i)] réel t ;*). Par exemple, l'effet de l'élaboration de la déclaration suivante :

```
ent x ;
```

peut être d'initialiser la variable avec la valeur entière par défaut. En pratique ceci peut être réalisé un peu différemment.

séquence
  si existétiquette de (*début*)
    alors
      sauvegarde
    finsi
  élaborer (série)
  δ.*début* ← δ.*série*
finséquence

Remarques :

1) Nous n'avons pas utilisé comme dans les noeuds précédents les actions *résultat* et *construirerésultat*, car il n'y a qu'une valeur délivrée. Ceci est exact si la condition suivante est réalisée :

- les zones des variables de blocs disjoints -en position collatérale à résultat non neutre- ne doivent pas se recouvrir.

Dans le cas contraire, si la valeur délivrée a sa représentation dans la zone des variables du bloc, cette zone des variables peut être écrasée par celle d'un autre bloc en position collatérale avec le premier. Pour éviter de perdre la représentation de la valeur, il faut en charger la représentation dans une ressource_résultat.

2) La génération conditionnelle du code de sauvegarde permet, comme dans le cas de l'instruction d'itération, de garder un contrôle statique sur les ressources de calcul et d'adressage, alors qu'à l'exécution plusieurs "chemins" sont possibles.

2.2 - (routine)

C'est donc sur ce noeud qu'est effectuée la gestion des environnements.

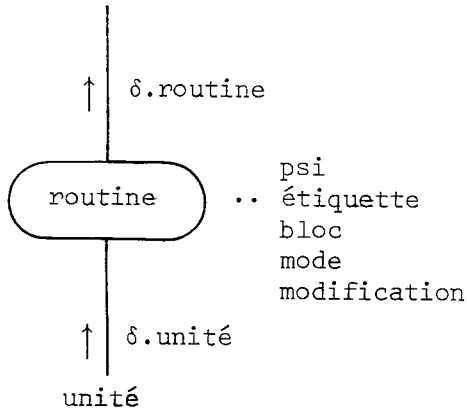
Pour pouvoir produire le code de la routine, il faut allouer et initialiser un environnement de compilation. Cet environnement est celui qui va servir à la compilation du texte de la routine.

Après cette création, le traitement réalise la génération séquentielle :

- du code de l'en-tête de la routine. Cet en-tête est composé d'une étiquette spéciale (point d'entrée) et d'un groupe d'instructions permettant : la mise en place des données de liaison, et la construction du vecteur d'accès,

- du code transmettant les parties dynamiques des valeurs des paramètres. Comme nous l'avons vu précédemment (cf. transmission de valeurs entre environnements,paramètres), seules les représentations des parties statiques des paramètres sont en place. Il faut donc recopier dans l'environnement de la routine les représentations des parties dynamiques si elles existent,
- du code correspondant au corps de la routine,
- du code transformant la valeur fournie en valeur-résultat,
- du code de fin de routine.

Ceci terminé, l'environnement de compilation courant est libéré et l'ancien environnement restauré.



décorations

psi : taille de la zone des variables déclarées dans la routine.

étiquette : étiquette réservée pour la routine, permettant de définir le point d'entrée de cette routine

bloc : ensemble des décorations représentant les caractéristiques statiques du bloc associé aux paramètres

mode : mode de la routine

modification : liste des modifications à effectuer sur la valeur.

Exemples : a)

```
proc ([,] ent t, réel r) ent :
   début
      réel x, ent y,
      :
      y
   fin
```

la décoration bloc : elle est associée au bloc :

([,] ent t, réel r)

la décoration psi : on tient compte des tailles statiques des paramètres t et r dans le calcul de la valeur de psi.

```
b) réel r := proc ent :
                début
                   ent j ; lire (j) ; j
                fin ;
```

la décoration modification comporte dans l'ordre un déprocédurage, qui permet d'activer la routine, suivi d'un élargissement, qui permet de transformer l'entier obtenu, après l'appel, en réel.

```
séquence
   descr δ
   Δenvcour ← nouvelenviron (psi de (routine) )
        co nouvel environnement de compilation co
   routineentête (étiquette de (routine) )
        co génération de l'en-tête de la routine et de la recopie du vecteur
           d'accès (display) co
   alloccopiepardyn (bloc de (routine) )
        co génération de la copie des parties dynamiques des paramètres co
   élaborer (unité)
   résultat (δ.unité)
        co génération de code pour transformer la valeur en valeur-résultat co
   routinefin
        co génération de la fin de la routine co
   Δenvcour ← libérerenviron
   δ ← construireroutine
   gmodif (δ, modification de (routine) )
   δ.routine ← δ
finséquence
```

```
fonction construireroutine = descr
   descr δ
   δ ← complétermode (δ, mode de (routine) )
        co indiquer le type "routine" dans le descripteur co
   faireroutine (δ)
finfonction

action alloccopiepardyn = (bloc bloc)
     co génération pour la copie des parties dynamiques des paramètres
        bloc : représente l'ensemble des paramètres de la procédure co

   pour tout identificateur idf ∈ bloc
   faire
        co idf correspond à la décoration identificateur du noeud (idf) co
      si dynamique (mode (idf))
      alors
        co le paramètre a des parties dynamiques co
         descr δ ;
        co descripteur de travail co
         δ ← descridf (idf)
        co initialisation du descripteur co
         affectation (δ, fant, dynamique)
        co génération de la copie des parties dynamiques co
      finsi
   finfaire
finaction

fonction modedecor = (identificateur i) type
     co
        retourner le mode contenu dans la décoration identificateur
     co
finfonction
```

2.3 - Conclusion -

La génération de code, et plus généralement la compilation, des instructions contrôlant la gestion des environnements ne pose pas de problèmes particuliers par rapport à Algol 60.

L'utilisation systèmatique de ces constructions dans le langage, due à l'orthogonalité de sa définition, peut donner aux programmes Algol 68 une forme parfois inattendue. Cette utilisation peut leur fournir non seulement une structure plus élégante, mais une plus grande efficacité à l'exécution.

3 - ACTIVATIONS -

Nous traitons ici les problèmes concernant les appels de routine Algol 68, les utilisations d'opérateurs et les appels aux fonctions d'entrée-sortie. Bien que ces trois constructions aient des traitements différents, elles ont un certain nombre de caractéristiques communes.

En effet :

- l'utilisation d'un opérateur défini dans le programme est équivalente à celle d'une routine ayant comme paramètres les opérandes de l'opérateur,
- l'appel d'une fonction d'entrée-sortie ne se différencie pas, dans sa forme externe, de celui d'une routine,
- l'ensemble de ces constructions entraîne un déroutement, à l'exécution, (sauf dans le cas des opérateurs standards) suivi d'un retour au point initial une fois le traitement terminé (ce qui les différencie des instructions de saut).

3.1 - Actions et fonctions spécifiques -

3.1.1 - Actions -

Le traitement de ces constructions nécessite la génération du code transmettant les représentations statiques des paramètres. Ceci est réalisé par l'action *paramètres* à laquelle on passe les descripteurs des valeurs à transmettre et le mode de la fonction à appeler.

```
action paramètres = (ensemble descr Δparamètres, type modeprimaire)
            co Δparamètres : ensemble des descripteurs représentant les
               paramètres effectifs,
               modeprimaire : mode de la routine que l'on veut activer co
            co cette action copie en place dans le nouvel environnement
               la partie statique des paramètres effectifs co
  descr δ, δbase
    co δ: descripteur temporaire servant à la copie
       δbase: descripteur représentant le nouvel environnement co
  allocbase (δbase, δtêtepile)
    co δtêtepile est le descripteur du pointeur sur la tête de pile co
```

```
pour tout descr δk ∈ Δparamètres
faire
    δ ← sélectionparamètre (δbase, k , modeprimaire)
        co met à jour le descripteur δ représentant la partie statique
           du k^ième paramètre à partir du mode du texte de routine co
    affectation (δ, δk, statique)
        co génération du code recopiant en place la partie statique co
finfaire
finaction
```

3.1.2 - Fonctions -

Trois fonctions sont utilisées dans le traitement de ces noeuds.

```
fonction résultat = (type modeprim) type
    co
       délivrer le mode du résultat d'une procédure dont le mode est
       donné par modeprim
    co
finfonction
```

```
fonction standard = (descr δ) bool
    co
      Suivant que le descripteur δ représente un objet du prélude standard
      la valeur "vrai" ou la valeur "faux" est retournée.
    co
finfonction
```

```
fonction sélectionparamètre = (descr δbase, ent k, type modeproc) descr
    co
       δbase : descripteur représentant la base de l'environnement de la
               routine à activer,
          k : numéro du paramètre,
    modeproc : mode de la routine à activer.
        allouer et initialiser un descripteur pour qu'il représente la partie
        statique (dans l'environnement de la routine à activer) du paramètre.
        Ce descripteur, une fois initialisé, à partir du mode de la routine
        et du descripteur δbase, est retourné comme résultat
    co
finfonction
```

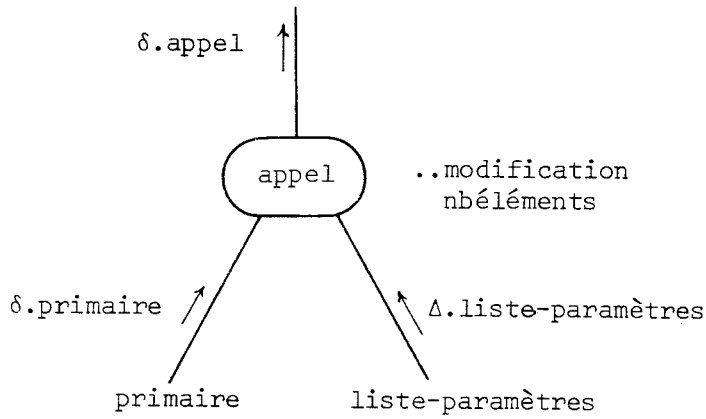
3.2 - (appel)

Il s'agit de l'instruction d'appel d'une routine avec paramètres. Dans le langage, les appels de routine sans paramètre sont définis comme des modifications de déprocédurage sur la valeur routine.

Exemple :

```
début
   ent x,y, proc p = ent : début
                              ent i ;
                              ... ; i
                           fin ;
        y:= x + p
fin
```

Si l'on considère l'expression $x + p$, la valeur entière de l'opérande droit est obtenue à partir de la variable p par déprocédurage.



décorations modification : liste de modifications appliquées à la valeur délivrée par l'appel
nbéléments : nombre de paramètres.

Les opérandes de ce noeud sont :

- l'arbre associé au "primaire" qui va permettre à l'exécution de connaître la routine à appeler.

Exemple :

Soit le programme :

```
début
      proc p = (ent) neutre : début ..... fin,
           q = (ent) neutre : début ..... fin ;
   si (bool b ; lire (b) ; b)
        alors p sinon q fsi (4)
fin
```

- l'ensemble des arbres correspondant aux arguments.
Cet ensemble est organisé sous forme de liste explicite, bien que les arguments (paramètres effectifs) s'évaluent collatéralement en Algol 68. La structure de liste facilite, en effet, l'association paramètres-arguments lors de l'appel, (une liste permet de structurer un ensemble ordonné fini), sans interdire l'évaluation collatérale (que l'on peut associer à chaque sous-liste). La transmission des valeurs des arguments dans le nouvel environnement doit respecter cet ordre.

Pour effectuer le traitement associé aux arguments de l'appel, on construit un ensemble ordonné de descripteurs Δ décrivant des arguments conformément à l'ordre de la liste. Cet ensemble Δ est construit par concaténation (notée +) des descripteurs des paramètres, en utilisant des règles semblables aux règles de dépendance , dans les systèmes d'évaluation d'attributs.

Le traitement consiste à engendrer collatéralement :

- le code évaluant le primaire,
- le code évaluant chaque argument (ce qui synthétise l'ensemble Δ lorsque la génération pour tous les arguments est terminée).

Ceci étant fait la routine est appelée en engendrant :

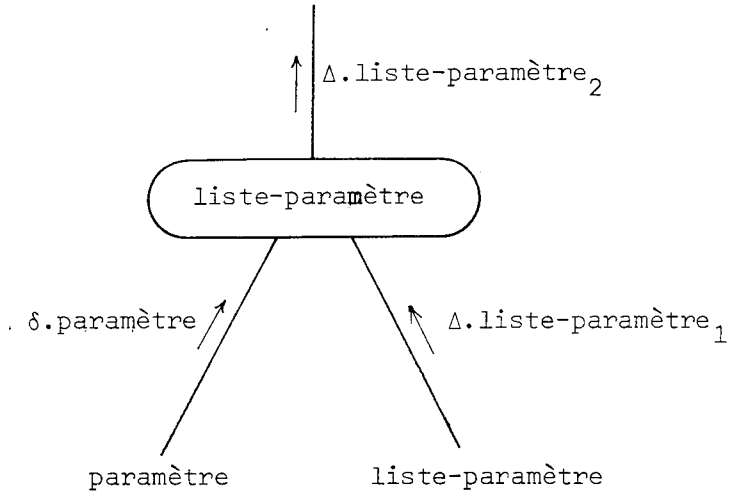
- la copie, des représentations statiques des valeurs des arguments, dans l'emplacement de la zone des variables réservé au paramètre associé à cet argument,
- l'appel à la routine correspondant au primaire.

La routine pouvant délivrer un résultat, du code est ensuite produit pour rendre cette valeur accessible depuis l'environnement de l'appel.

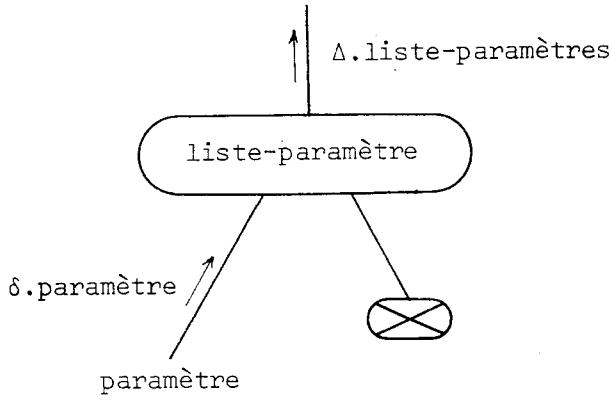
Ceci se fait en deux étapes :

- d'abord le transfert de la représentation statique de la valeur dans la zone courante d'évaluation (gestion statique),
- ensuite le transfert de la représentation dynamique (si elle existe) de la valeur dans la zone dynamique courante si elle n'est point accessible depuis cet environnement (c'est-à-dire si elle est locale à la routine appelée).

séquence
  descr δres
  collatéral
    élaborer (primaire)
    élaborer (liste-paramètre)
      co traitement de la liste des arguments avec synthétisation de Δ.*liste-paramètre co*
  fincollatéral
  paramètres (Δ.liste-paramètre, modedescr (δ.primaire))
      co génération du code rangeant les parties statiques des arguments en place dans la zone des variables créée dans le nouvel environnement co
  activerroutine (δ.primaire)
      co activation de la routine co
  construireappel (δres, modedescr (δ.primaire))
  gmodif (δres, modification de (*appel*))
      co traitement des modifications co
  δ.*appel* ← δres
finséquence



```
séquence
   collatéral
      élaborer (paramètre)
      élaborer (liste-paramètre)
        co traitement du reste de la liste des arguments co
   fincollatéral
   Δ.liste-paramètre2 ← δ.paramètre + Δ.liste paramètre1
finséquence
```




```
séquence
   élaborer (paramètre)
   Δ.liste-paramètre ← δ.indice
finséquence
```

```
action construireappel = (descr δres, type modeprimaire)
   descr δ
     co compléter le descripteur δ à partir du mode du primaire pour qu'il
        représente la valeur résultat de l'appel (en ressource résultat).
        Transférer cette valeur dans la zone courante d'évaluation en modi-
        fiant le descripteur δres de manière à ce qu'il représente cette
        valeur transférée
     co
   δ ← complétermode (δ, résultat (modeprimaire))
   setres (δ, δres)
     co δres représente l'emplacement dans la pile statique d'évaluation
        contenant la valeur co
   si dynamique (modedescr (δres))
      alors
           co la valeur résultat est rendue accessible dans l'environ courant
              par copie éventuelle de sa partie dynamique (cf système à
              l'exécution) co
      copiedynamique (δres)
           co génération de la copie de la valeur dynamique si celle-ci ne
              se trouve pas dans l'environnement actif courant co
   finsi
finaction
```

3.3 - (opérateur)

L'opérateur a été identifié préalablement pendant la construction de l'arbre abstrait décoré.

Le traitement consiste à engendrer le code permettant:

- l'élaboration collatérale des deux opérandes (un seul si l'opérateur est unaire)

- 1) dans le cas des opérateurs standard , le code en ligne réalisant, sur les opérandes, l'opération demandée.

  2) dans le cas des opérateurs non standard , le code réalisant le même traitement que celui des appels, c'est-à-dire :

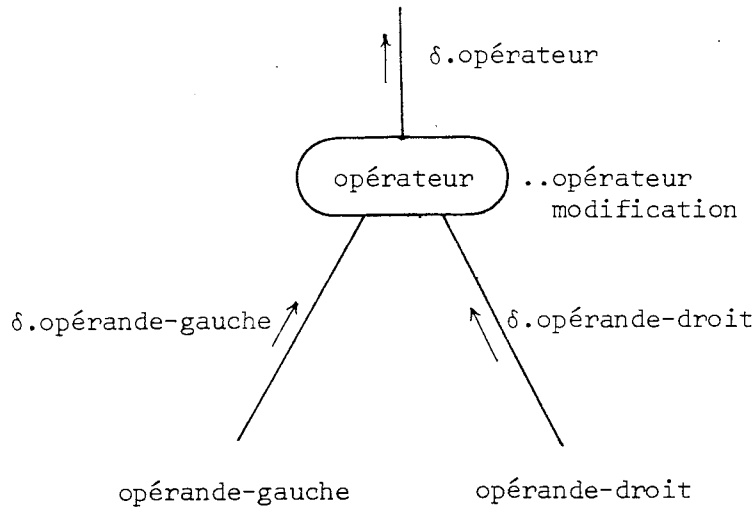
  - la copie des représentations statiques des opérandes,
  - l'appel à la routine associée à l'opérateur,
  - le transfert de la représentation statique du résultat,
  - le transfert de la représentation dynamique (si besoin est).

Exemple :

```
début
   [1:4] ent t, u ;
      :
   op + = ([] ent, [] ent) [] ent :
      début
         ...
      fin ;
      ...
   u := t + u ;
      ...
fin
```

Le cas des opérateurs binaires a été séparé de celui des opérateurs unaires.

3.3.1 - Opérateur binaire -



Décorations

opérateur : ensemble des caractéristiques statiques de l'opérateur :
- paramètres de production de code, s'il est standard
- caractéristiques du texte de routine, s'il n'est pas standard

modification : liste des modifications appliquées sur le résultat de l'opération.

séquence

descr δop, δ

co descripteurs pour représenter l'objet associé à l'opérateur co

δop ← *descrop (opérateur de* (opérateur))

co δop *représente l'objet associé à l'opérateur co*

collatéral

élaborer (opérande-gauche)

élaborer (opérande-droit)

co génération du code pour les opérandes co

fincollatéral

*si standard (*δop)

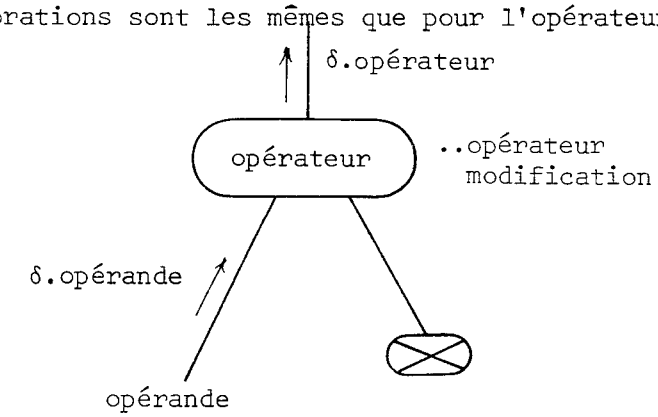
*alors opstandbin (*δ.*opérande-gauche,* δ.*opérande-droit,* δop, δ)

co génération en ligne du code de l'opération co

```
    sinon
        paramopbin (δ.opérande-gauche, δ.opérande-droit, modedescr (δop))
            co génération du code rangeant les parties statiques des opérandes
               en place dans la zone des variables de l'environnement de la
               routine à appeler co
        activerroutine (δop)
            co génération du code appelant la routine associée à l'opérateur co
        construireopérateur (δ, modedescr (δop))
    finsi
    gmodif (δ, modification de (opérateur) )
    δ.opérateur ← δ
finséquence
```

3.3.2 - Opérateur unaire

Les décorations sont les mêmes que pour l'opérateur binaire




```
séquence
    descr δop, δ
    δop ← descrop (opérateur de (opérateur) )
        co δop représente l'objet associé à l'opérateur co
    élaborer (opérande)
    si    standard (δop)
    alors opstandun (opérande, δop, δ)
    sinon
        paramopun (δ.opérande, modedescr (δop))
            co génération du code rangeant, la partie statique de l'opérande, en
               place dans l'environnement de la routine à appeler co
        activerroutine (δop)
        construireopérateur (δ, modedescr (δop))
    finsi
    gmodif (δ, modification de (opérateur) )
    δ.opérateur ← δ
finséquence
```

```
action construireopérateur = (descr δres, type mode)
   descr δ
     co     δ : descripteur pour représenter le résultat de l'opération
        mode : mode de l'opérateur
        l'opérateur étant non-standard, le résultat est une valeur-résultat co
   δ ← complétermode (résultatmode (mode))
     co δ représente la valeur ayant les caractéristiques de celle du résultat
        de l'opération co
   setres (δ, δres)
   si dynamique (modedescr (δres)
      alors
         copiedynamique (δres)
           co génération de la copie de la valeur dynamique si celle-ci ne se
              trouve pas dans l'environnement actif courant co
      finsi
finaction

action paramopbin = (descr δg, δd, type mode)
     co δg et δd sont les descripteurs représentant les opérandes
        mode : mode de l'opérateur co
   paramètres (δg + δd, mode)
     co génération de la copie en place des parties statiques des opérandes co
finaction

action paramopun = (descr δ, type mode)
   paramètres (δ, mode)
finaction

action opstandbin = (descr δg, δd, δop, δ) descr
     co δg et δd sont les descripteurs représentant les opérandes
        δop : le descripteur représentant l'opérateur
          δ : descripteur pour le résultat de l'opération
     co
   cas ent typeop (δop)
   dans
     co suivant l'opérateur standard génération du code de l'opération à
        effectuer. Cette génération utilise des ressources-machine. Le
        résultat de l'opération est dans une ressource-machine. Le
        descripteur δ est initialisé en conséquence co
```

```
    fincas
      co compléter le descripteur δ pour qu'il représente une valeur ayant
         les caractéristiques du résultat de l'opérateur co
        complétermode (δ, résultat (modedescr (δop)))
finaction

action opstandun = (descr δun, δop, δ) descr
     co cf opstandbin co
finaction

fonction typeop = (descr δ) ent
     co
        suivant l'opérateur représenté par le descripteur δ, délivrer une
        valeur entière caractéristique de cet opérateur
     co
finfonction
```

3.4 - (entrée-sortie)

Ce noeud a deux opérandes :

- l'arbre associé au fichier sur lequel doit être réalisé l'entrée-sortie
- un ensemble ordonné d'arbres associés aux éléments à entrer ou à sortir. Cet ensemble est représenté sous une forme explicite de liste. La liste respecte l'ordre spécifié par le programme.

Le traitement consiste à engendrer :

- le code correspondant à l'évaluation de l'opérande fichier
- le code correspondant à chaque élément à entrer ou à sortir. S'il s'agit d'une valeur composée (structure ou tableau) ce code la déploie ("straightening") en valeurs simples avant de réaliser l'entrée-sortie. S'il s'agit d'une valeur format, le code la transmet, ainsi que le fichier, à la routine d'évaluation des formats. Le code produit doit entrer ou sortir les valeurs dans l'ordre précisé par le programmeur. Le deuxième opérande est donc une liste traitée séquentiellement.

Remarque :

D'autres valeurs composées (de mode union) peuvent être entrées ou sorties de la même manière sous certaines conditions. Les constituants de l'union doivent être susceptibles d'une opération d'entrée-sortie.

Ainsi, ils ne peuvent être :

- en entrée, que de mode repère d'un mode de valeur éditable,
- en sortie, que du mode d'une valeur éditable.

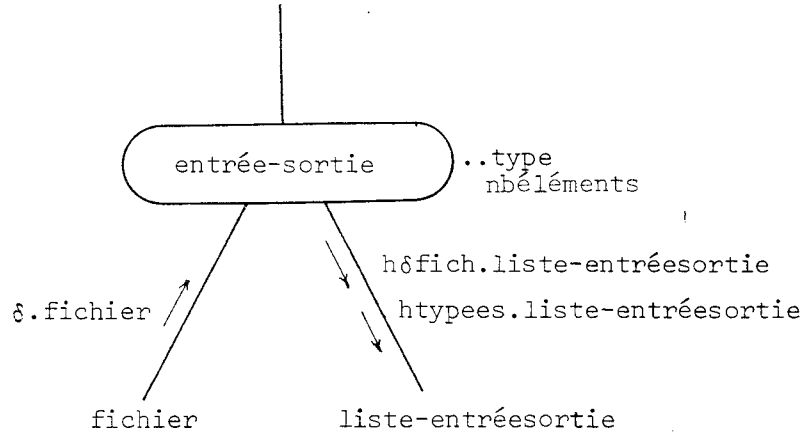
Exemples :

Soit le programme :

```
début
   ent e,
   union (rep ent, rep [] réel) u,
   union (ent, [] réel) v ;
   u := e ; v := 1 ;
   ...
   lire (u) ; co imprimer (u) co ;
   co lire (v) co ; imprimer (v) ;
   ...
fin
```

L'instruction lire (u) est sémantiquement correcte, par contre imprimer (u) n'est pas correct, la modification dérepérer ne pouvant être appliquée aux constituants de l'union. De même lire (v) n'est pas correct : v n'étant pas de mode repère de mode éditable (la variable v est dérepérée) ; par contre imprimer (v) est correct.

De telles situations peuvent paraître paradoxales au premier abord, et montrent les limites de l'orthogonalité dans la définition d'un langage aussi riche qu'Algol 68.



Décorations

type : indication du type de l'opération d'entrée-sortie à effectuer. On distingue 6 types distincts dans le langage Algol 68 :

- lecture sans format
- écriture sans format
- lecture avec format
- écriture avec format
- lecture binaire
- écriture binaire

Les entrées-sorties (avec ou sans format) sur les fichiers standards se ramènent aux cas précédents.

nbéléments : nombre d'éléments de la liste.

Informations héritées

Les informations représentant le type de l'entrée-sortie et le fichier sur lequel est effectué l'opération sont héritées sur l'opérande liste pour pouvoir engendrer le code nécessaire à l'opération d'entrée-sortie des éléments.

Les informations :

- hδfich.liste-entréesortie,
- htypees.liste-entréesortie, sont ainsi héritées.

séquence

élaborer (fichier)

co génération de code pour élaborer le paramètre fichier co

hδfich.liste-entréesortie ← δ.fichier

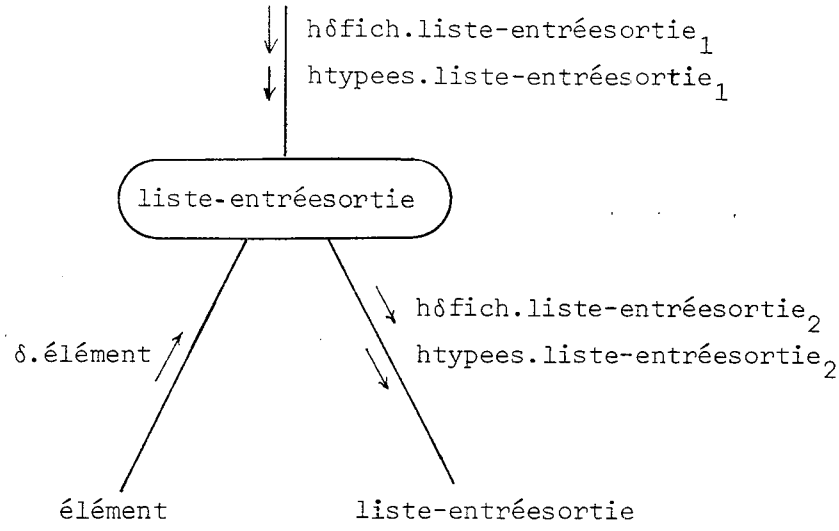
htypees.liste-entréesortie ← type de (entrée-sortie)

co informations héritées sur la liste d'entrées-sorties co

élaborer (liste-entréesortie)

co génération du code pour les opérations d'entrées-sorties co

finséquence



séquence

élaborer (élément)

co génération de code pour l'élaboration de l'élément que l'on veut entrer ou sortir co

entréesortie (δ.élément, hδfich.liste-entréesortie$_1$
htypees.liste-entréesortie$_1$)

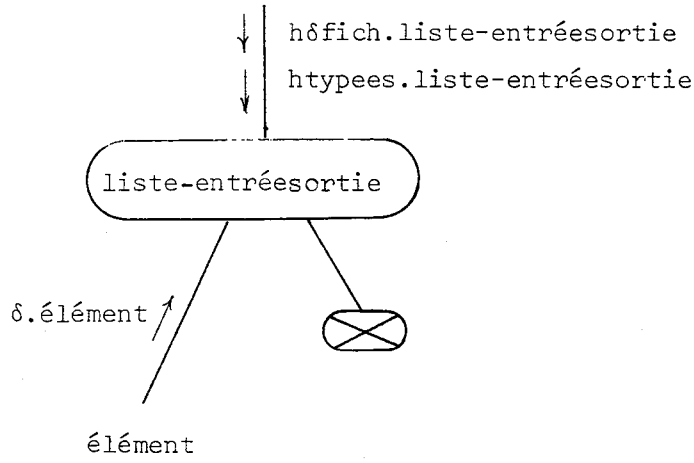
co génération de code pour l'opération d'entrée-sortie de la valeur représentée par δ.élément co

htypees.liste-entréesortie$_2$ ← htypees.liste-entréesortie$_1$

hδfich .liste-entréesortie$_2$ ← hδfich .liste-entréesortie$_1$

élaborer (liste-entréesortie)

finséquence



séquence

élaborer (élément)

entréesortie (δ.élément, hδfich.liste-entréesortie,
htypees.liste-entréesortie)

finséquence

```
action entréesortie = (descr δélem, δfich, ent typees)
     co δélem  : descripteur de l'élément à entrer ou sortir
        δfich  : descripteur de l'objet fichier
        typees : type de l'entréesortie à effectuer (entrée ou sortie avec
                 ou sans format) co
   cas type modedescr (δélem) dans
      procédition :
        co routine d'édition dont le paramètre est δfich co
         paramètres (δfich, modedescr (δélem))
             co copie en place de la partie statique du paramètre fichier co
         activerroutine (δélem)
             co activation de la routine d'édition co,
      format :
        co objet de mode format co
         paramètres (δfich + δélem, modeformat)
             co copie partie statique du paramètres fichier co
         format
             co appel de la routine d'élaboration des formats co,
      sinon
        co objet dont la valeur doit être éditée ou inditée co
         déployer (δélem, δfich, typees)
             co génération de code pour le déploiement de la valeur et
                l'opération d'entrée-sortie co
   fincas
finaction

action déployer = (descr δélem, δfich, ent typees)
     co  δélem  : descripteur de l'élément
         δfich  : descripteur de l'objet fichier
         typees : type de l'entréesortie co
     co génération, en fonction du mode, du code permettant :
        - de parcourir la valeur si elle est arborescente
        - d'appeler pour chaque valeur simple (obtenue éventuellement après
          déploiement) la routine de bibliothèque permettant d'effectuer sur la
          valeur l'opération d'entrée-sortie
     co
finaction
```

modèle format

co génération de l'appel de la routine d'évaluation du format co

(appelbiblio, format,,)

finmodèle

3.5 - Conclusion -

Avec le noeud (entrée-sortie) , on termine la description des noeuds associés aux appels et à l'utilisation d'opérateurs.

On peut remarquer, au niveau de la compilation, que (par rapport à Algol 60) la génération de code ne pose de problèmes nouveaux que :

- pour les opérateurs (distinction entre opérateurs standards et non standards),
- et pour les entrées-sorties (les valeurs simples ne sont plus les seules concernées).

Les problèmes posés par l'identification des opérateurs ont été examinés lors de la construction de l'arbre abstrait décoré.

Au niveau de l'utilisation du langage, l'introduction de possibilités de définition de nouveaux opérateurs et les possibilités d'effectuer des entrées-sorties sur des valeurs composées facilitent l'écriture des programmes.

4 - GENERATEURS ET DECLARATIONS ALGOL 68 -

Dans ce chapitre, nous traitons les problèmes de génération de code pour les constructions suivantes du langage :

- déclarations de mode
- générateurs
- déclarations de variable
- déclarations d'identité
- déclarations d'opérateur
- étiquettes.

L'allocation d'espace sous-entendue par certaines de ces déclarations est un problème fondamental. Il va nécessiter pour les déclarations de mode, les générateurs et les déclarations de variable (c'est-à-dire les constructions utilisant des déclareurs en position *effective*) la mise en place d'un cadre,spécifique du déclareur utilisé par ces constructions.

Dans la deuxième partie, nous passons en revue le traitement associé à chacune de ces constructions.

4.1 - Traitement des constructions utilisant des déclareurs effectifs -

D'une manière générale, on peut distinguer deux sortes de traitement suivant que le déclareur, utilisé dans le programme, décrit des valeurs avec ou sans parties dynamiques. Par extension, nous parlerons, dans les pages suivantes, de déclareurs avec ou sans parties dynamiques.

Dans le cas des déclareurs sans parties dynamiques, le mode fournit toutes les caractéristiques de la représentation des valeurs associées. De ce fait, les problèmes d'allocation d'espace sont grandement simplifiés.

Il n'en est pas de même pour les déclareurs à parties dynamiques. En effet, l'ensemble des caractéristiques de la représentation des valeurs de ce mode n'est connu qu'au moment de l'exécution. Du code est alors produit pour déterminer et exploiter ces caractéristiques.

4.1.1 - Allocation d'espace pour les objets à parties dynamiques -

Par de tels objets nous entendons ceux dont le déclareur *effectif* associé contient des tableaux pour lesquels les bornes doivent être précisées (déclareur à parties dynamiques).

Exemples de déclareurs à parties dynamiques :

[1:n] ent
struct ([x:y+3] [1:10] ent a, réel b)

Pendant l'exécution, de tels objets vont demander dynamiquement des allocations d'espace.

En Algol 68, un objet quelconque ne peut être créé que via un générateur, qui peut être explicite ou sous-entendu (déclarations de variables). Ce générateur est constitué d'un déclareur de mode pouvant :

- soit fournir à lui seul toutes les informations concernant le mode effectif :
  Exemples de déclareurs de mode :

  [1:n] ent
  struct (réel a, [1:10] bool b)

- soit utiliser des indicateurs de mode pour fournir ces mêmes informations :
  Exemples de déclareurs utilisant des indicateurs :

  m1
  struct (réel a, m2 b)
  avec { *mode m1 = [1:n] ent*
  mode m2 = [1:10] bool

4.1.1.1 - Il faut bien saisir la différence entre ces deux façons de faire car elle est fondamentale pour la suite.

- A l'exécution, les deux exemples précédents, placés au même endroit dans le même programme, doivent produire le même effet. Ceci signifie que, dans le deuxième cas, les bornes des tableaux et la taille de l'objet correspondant ne doivent pas être calculées lors de l'élaboration des déclarations de mode mais lors de celle des générateurs. En conséquence, lors de l'exécution, une déclaration de mode ne nécessite aucune élaboration.

- L'élaboration d'un générateur (c'est-à-dire l'allocation et l'initialisation de l'espace demandé), nécessite de connaître le déclareur (y compris les bornes) associé à tout indicateur de mode utilisé dans ce générateur. En conséquence, il faut disposer, à cet instant, du code évaluant les bornes des tableaux contenus dans le déclareur de chaque indicateur utilisé.

Pendant la phase de génération, ce code n'a pu être produit que lors du traitement de la déclaration de mode (déclaration de l'indicateur). Plutôt que de dupliquer ce code chaque fois que le traitement d'un générateur le nécessite, nous en avons fait un texte de routine. Chaque routine est associée à l'indicateur correspondant et peut, de ce fait, être appelée au niveau de tout générateur utilisant cet indicateur en position effective.

- A l'exécution, il ne suffit pas de pouvoir appeler une routine qui évalue les bornes des tableaux, il faut aussi que cette routine transmette les résultats de ces évaluations pour pouvoir, au niveau du générateur, faire l'allocation et l'initialisation de l'espace demandé. C'est pourquoi la routine appelée va sauvegarder temporairement les résultats qu'elle a obtenus (notamment les bornes). Ces informations, exploitées par le code produit pour le générateur, vont permettre l'allocation et l'initialisation de l'espace.

- Cette zone de sauvegarde (de transfert d'informations) doit être connue aussi bien par la routine d'évaluation des bornes (c'est elle qui la remplit) que par le code produit pour le générateur (c'est lui qui l'exploite). Cette zone est réservée (dans la zone d'évaluation) lors du traitement du générateur (sa taille est connue grâce à la décoration transfert) et son accès est transmis comme paramètre à la routine d'évaluation.

4.1.1.2 - Examinons le traitement à réaliser sur un générateur à parties dynamiques.

a) Si le déclareur associé est réduit à un indicateur il suffit, d'après ce que nous venons de dire, d'engendrer un appel à la routine associée au déclareur de cet indicateur (en lui passant l'accès à la zone de sauvegarde).

Exemple :

```
début
    ent a,b ;
    ...
    mode m1 = [a:b] réel ;
    ...
    m1 x ;
    ...
fin
```

La déclaration *m1 x* utilise implicitement un générateur (cette déclaration est équivalente à *rep m1 x = loc m1*).
Le déclareur de ce générateur est réduit à l'indicateur *m1*.
L'appel de la routine associée à *m1* permet d'obtenir les informations nécessaires à l'allocation et l'initialisation de l'espace.

b) Si le déclareur associé ne se réduit pas à un indicateur, nous utilisons un procédé semblable pour obtenir ces mêmes informations.

Durant la phase de génération, une routine est produite pour ce déclareur. Cette routine est exactement la même que celle qui serait produite si le même déclareur était le déclareur effectif d'une déclaration de mode.

Il suffit alors pendant le traitement du générateur de produire un appel à cette routine de la même manière que précédemment.

En remarquant qu'il n'existe alors qu'un seul appel à cette routine, on peut se demander s'il n'est pas préférable de produire "en place" la séquence de code correspondante.

Cette méthode présente deux avantages :

- par augmentation du nombre de routines, elle permet de diminuer le volume du code produit pour toute routine contenant de telles déclarations. On obtient ainsi un programme objet plus modulaire. Précisons que le volume total du code engendré n'est absolument pas diminué. Il est même augmenté du code d'appel de la routine.

- le fait d'avoir une même représentation abstraite pour des traitements informatiques ayant la même sémantique,mais des représentations concrètes différentes, est actuellement considéré comme étant essentiel.

Citons, par exemple, les essais de traduction de plusieurs langages de représentations externes différentes vers un même langage noyau. La méthode que nous proposons abonde dans ce sens puisqu'elle permet d'avoir toujours le même traitement pour ces constructions (qui produisent le même effet à l'exécution quelles que soient leurs représentations externes).

4.1.2 - Génération et utilisation de la routine associée à un déclareur -

4.1.2.1 - Génération -

Trois noeuds vont avoir comme fils, l'arbre associé à leur déclareur. Il s'agit des noeuds :

(déclaration-d-mode) , (d-générateur) , (déclaration-d-variable)

Pour ces trois cas, au fur et à mesure du parcours de leur arbre, nous produisons le code de la routine associée.
Pour ce faire, nous utilisons une action de parcours spéciale (action *élaborerdéclareur*) appelée depuis ces trois noeuds avec comme paramètres l'arbre associé au déclareur et les informations nécessaires pour produire un en-tête et une fin de routine.

Une fois ce parcours terminé, un descripteur, représentant la routine associée au déclareur, est synthétisé vers celui de ces noeuds d'où s'est fait l'appel à élaborerdéclareur.

Remarque :
Rappelons que l'arbre associé au déclareur n'a des branches que pour les sous-déclareurs ayant des parties dynamiques (et donc des bornes à évaluer).

Exemple :

Si le mode est

[x:y] struct (ent a, [z:t] réel b)

L'arbre associé au déclareur n'aura pas de branche pour le déclareur *ent* du champ *a* ni pour le déclareur *réel* des éléments du tableau *b*.

```
action élaborerdéclareur = (arbre déclareureff, étiquette étiquette, ent psi)
                                descr
    co cette action permet de produire, à partir du sous-arbre abstrait
       déclareureff représentant le déclareur effectif, le code dont l'éla-
       boration permet d'évaluer les bornes des tableaux contenus dans ce
       déclareur et de sauvegarder sur la pile ces valeurs.
       Etiquette est l'étiquette de début de la routine associée au déclareur
       effectif co
descr δrout
    co descripteur pour le texte de routine co
Δenvcour← nouvelenviron (psi)
    co nouvel environnement de compilation co
routineentête (étiquette)
    co génération de l'en-tête de la routine et de la recopie du vecteur
       d'accès (display) co
δtransf.déclareur ← modifieraccès (baseparam, Δenvcour)
    co initialisation du descripteur décrivant l'accès à la zone de transfert.
       Cet accès est le paramètre de la routine du déclareur co
élaborer (déclareureff)
    co génération du code pour le déclareur effectif et initialisation de la
       zone de transfert co
routinefin
    co génération de la fin de routine co
Δenvcour ← libérerenviron
    co libération de l'environ de compilation co
δ.déclareur ← faireroutine (δrout)
    co synthèse d'un descripteur de routine co
finaction
```

La génération du code formant le texte de la routine associée au déclareur se fait en parcourant l'arbre qui n'est formé que des noeuds

(rang) , (structure) ou (indicateur)

(en ne tenant pas compte des sous-arbres associés aux bornes).

Le langage permet la "mise en facteur" des déclareurs des champs d'une structure (ex : *struct ([1:10] ent a,b,c)*).
Nous retrouvons cette particularité au niveau des noeuds de l'arbre du déclareur, avec la décoration duplication qui indique à combien de sélecteurs s'applique le déclareur.

La taille de la zone de transfert est calculée statiquement en fonction du mode et ne dépend pas de telles "mises en facteur" aussi est-il nécessaire d'engendrer du code dupliquant les informations rangées dans cette zone (par l'évaluation des bornes d'un tel sous-déclareur).

En fin de traitement de chaque noeud de l'arbre du déclareur, l'action *dupliquer* prend en charge la génération de ces duplications.

Remarque :
De manière générale, lorsqu'il y a "mise en facteur" d'un déclareur (ou d'un sous-déclareur) la définition du langage interdit la "défactorisation".

Exemple :

```
début
   réel x ;
   [1 : (ent i ; lire (i) ; i)] réel t,u ;
   lire (x) ;
   ...
fin
```

La déclaration

```
[1: (ent i ; lire (i) ; i )] réel t,u ;
```

pourrait être défactorisée de deux manières différentes :

1) en séparant les déclarations par une virgule :

```
[1: (ent i ; lire (i) ; i )] réel t , [1: (ent i ; lire (i) ; i)] réel u ;
```

2) ou en séparant les déclarations par un point virgule :

```
[1: (ent i ; lire (i) ; i)] réel t ; [1:(ent i ; lire (i) ; i)] réel u ;
```

Quelle que soit la défactorisation adoptée, le Rapport devrait définir précisément les règles d'application de la défactorisation car les répercussions de ce genre d'opérations sont importantes.

Par exemple, l'exécution de la déclaration défactorisée va nécessiter deux lectures d'entiers, ce qui oblige le programme à mettre deux entiers dans ses données, sinon la deuxième lecture *lire (i)* va récupérer la valeur réelle prévue pour x.

La deuxième défactorisation ne peut pas être retenue car elle ne respecte pas la sémantique du texte source. En effet, elle transforme une élaboration collatérale, exprimée par la virgule, en une élaboration séquentielle.

La première solution pose le problème suivant :
- les deux valeurs à mettre dans les données doivent-elles être les mêmes ?

Si ce n'est pas le cas, le programmeur ne peut pas prévoir le nombre d'éléments de chacun des deux tableaux. En effet, les deux déclarations étant en position collatérale, il n'est pas possible de connaître l'ordre dans lequel elles seront élaborées.
Si, par contre, les deux valeurs doivent être les mêmes, il faut alors produire du code vérifiant cette condition. Ce serait payer bien cher une simple facilité d'écriture.

Quelle que soit l'opération de défactorisation choisie, la sémantique de la construction initiale n'est pas conforme à celle de la construction obtenue. C'est pourquoi la défactorisation n'est pas définie dans le langage. Les constructions factorisées ne sont donc à élaborer qu'une seule fois.

```
action dupliquer = (descr δadup, ent nbdup, longtransfert)
     co δadup est le descripteur représentant la zone à dupliquer
        nbdup est le nombre de duplications
        longtransfert est la longueur de la zone à dupliquer co
   ent i ← 1
   descrδ
     co comment descripteur temporaire pour recopie co
   tantque i < nbdup
   faire
        δ ← modifieraccès (longtransfert, δadup)
     co δ représente l'emplacement où il faut recopier co
        copiercontigu (δadup, δ)
     co produit la recopie depuis l'emplacement représenté par δadup vers
        l'emplacement représenté par δ co
        i ← i+1
     co duplication suivante co
   finfaire
finaction
```

4.1.2.2 - Utilisation -

Chaque routine associée à un déclareur effectif a comme paramètre une zone de transfert dans laquelle sont rangés les résultats des évaluations. Il faut donc prévoir l'allocation (en zone d'évaluation) de cette zone avant de produire l'appel. Cet appel, quelle que soit la routine, respecte toujours les mêmes conventions. La génération de ces appels est réalisée par l'action *routinedécla* qui a comme paramètres :

- le descripteur représentant la routine à appeler
- le descripteur de la zone de transfert associée.

```
action routinedécla = (descr δroutine, δtransf)
    co δroutine : descripteur représentant la routine associée au déclareur
       effectif
       δtransf  : descripteur de la zone de transfert associée co
  descr δparamètre
  allocbase (δparamètre, δtêtepile)
  δparamètre ← modifieraccès (baseparam, δparamètre)
    co le descripteur δparamètre représente l'accès au premier paramètre
       dans un nouvel environ à l'exécution - ce descripteur tient compte
       des données de liaison dans un nouvel environ à créer au sommet de
       pile co
  copieaccès (δparamètre, δtransf)
    co génération du code de rangement de l'accès
       (à la zone de transfert effective) dans l'emplacement du premier
       paramètre co
  activerroutine (δroutine)
    co génération de l'appel de la routine associée co
finaction
```

4.1.3 - Traitement des noeuds de l'arbre associé à un déclareur

Le noeud (rang) correspond à un (sous-) déclareur de type tableau. Ses fils sont :

- une liste représentant l'ensemble des arbres de ses bornes. Chaque noeud (qui correspond à une paire de bornes) de cet ensemble a comme décoration le déplacement de l'enregistrement, dans la zone de transfert, où doivent être rangées les bornes de la dimension correspondante.
- l'arbre du déclareur de ses éléments (vide s'ils n'ont pas de parties dynamiques).

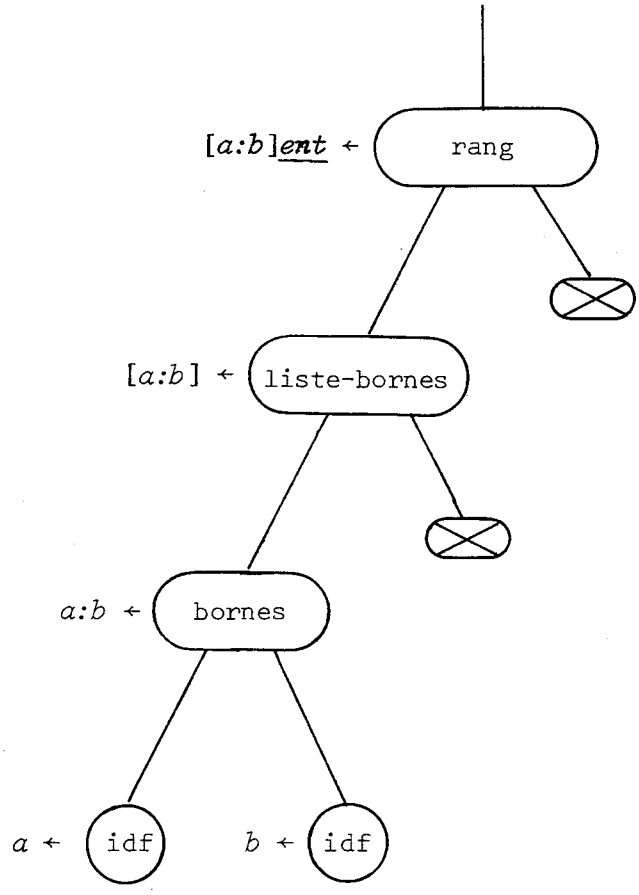
Le noeud (structure) correspond à un (sous-) déclareur de type structure. Son fils est une liste représentant l'ensemble des arbres associés aux déclareurs à parties dynamiques de ses champs. Chaque noeud (qui correspond à un champ dynamique) de cet ensemble a comme décoration le déplacement de l'enregistrement (dans la zone de transfert) où doivent être rangées les bornes du déclareur du champ correspondant.

Le noeud (indicateur) correspond à l'utilisation (en tant que sous-déclareur) d'un indicateur dont le déclareur associé a des parties dynamiques (une routine a donc été produite lors du traitement de la déclaration de cet indicateur).

Exemple : Considérons le programme suivant :

```
début
   ent a, b, c, d ;
   ...
   mode m1 = [a:b] ent ;
   mode m2 = struct (réel cs,ds) ;
   struct ([a:c] m1 tx, ty, tz, réel rx, m2 sx, sy, m1 tu) str ;
   ...
fin
```

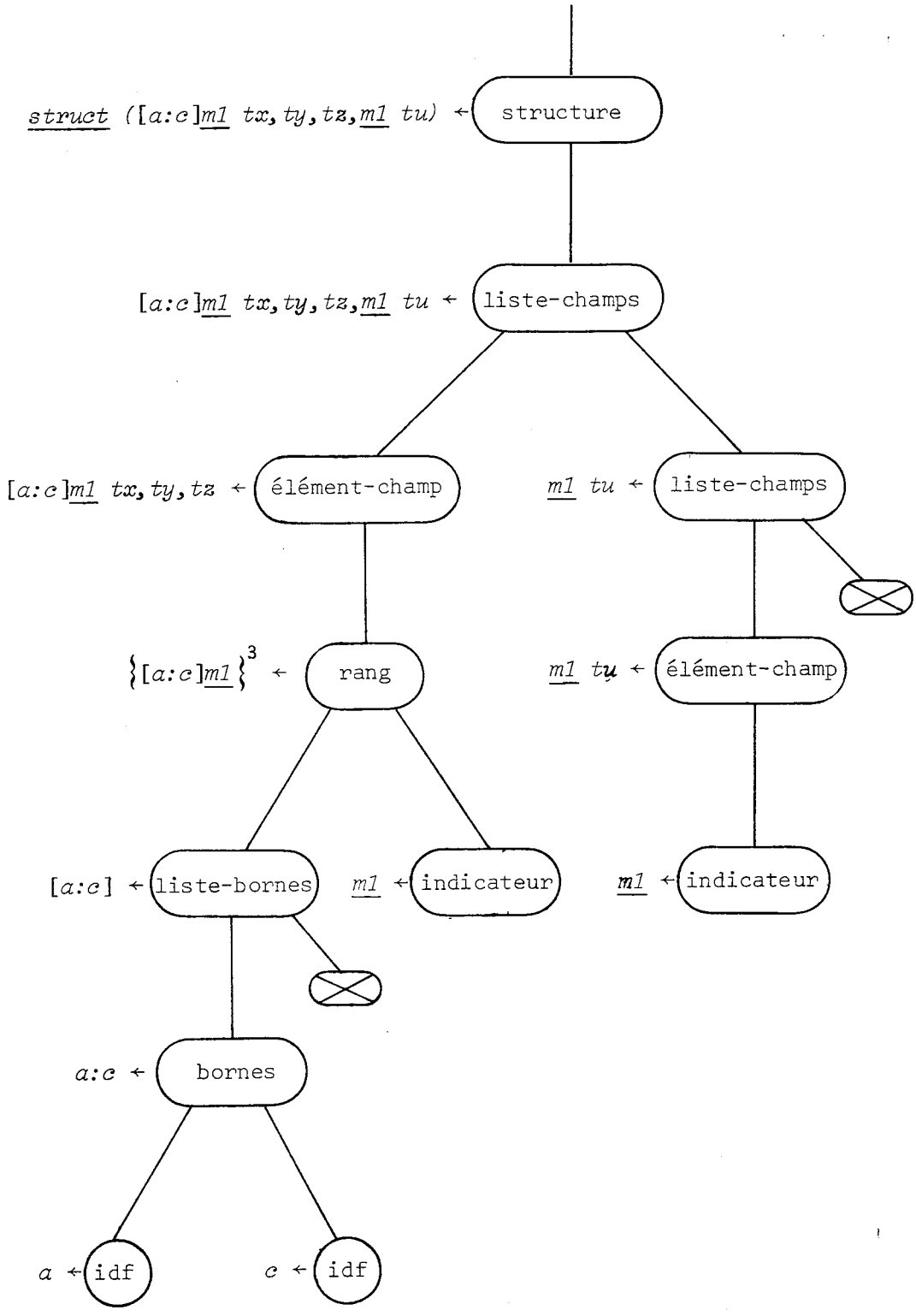
L'arbre abstrait du déclareur associé à l'indicateur *m1* se représente par :



Le symbole ← associe à un noeud la chaîne source correspondante.

Remarquons que le noeud (rang) n'a pas de fils droit car le mode des éléments du tableau n'est pas dynamique.

L'arbre abstrait associé au déclareur de *str* se représente par :

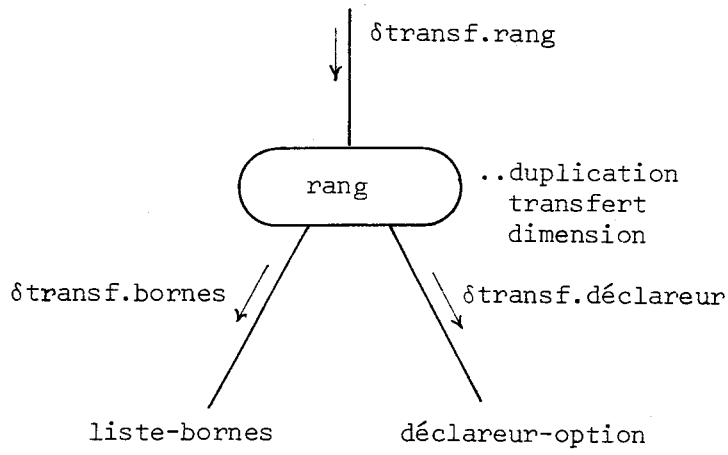


Remarques :

- les champs *réel rx* et *m2 sx, sy* n'apparaissent pas dans l'arbre car les déclareurs correspondants ne sont pas dynamiques.

- $\{[a:c]\ m1\}^3$ indique que le déclareur $[a:c]\ m1$ est mis en facteur pour les trois champs *tx, ty* et *tz*.

4.1.3.1 - (rang)



décorations

duplication = nombre de fois où le déclareur associé est "mis en facteur"

transfert = longueur de la zone de transfert à dupliquer

dimension = nombre de dimensions du tableau

information héritée

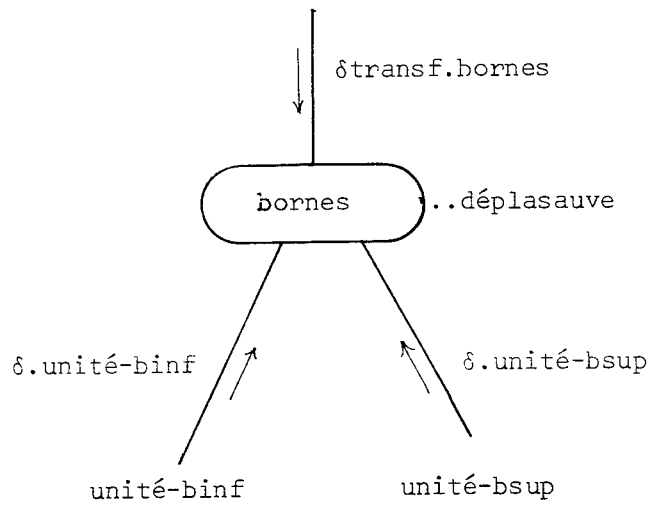
δtransf.rang = descripteur représentant l'accès courant à la zone de transfert

informations à hériter

δtransf.bornes = descripteur représentant l'accès courant à la zone de transfert pour la sauvegarde des bornes

δtransf.déclareur = descripteur représentant l'accès courant à la zone de transfert pour la sauvegarde des informations (si besoin est) associées au déclareur des éléments du tableau.

```
collatéral
   pour tout arbre bornes ∈ liste-bornes
   faire
      séquence
         δtransf.bornes ← δtransf.rang
         élaborer (bornes)
            co génération du code permettant l'évaluation des bornes et la
               sauvegarde de leur valeur dans la zone de transfert co
      finséquence
   finfaire
   si nonnil (déclareur-option) alors
      séquence
         δtransf.déclareur ← modifieraccès (longsauve (dimension de (rang) ),
                                                δtransf.rang)
            co descripteur représentant l'accès à la zone de transfert pour
               l'opérande déclareur co
         élaborer (déclareur-option)
            co traitement du déclareur des éléments dynamiques co
      finséquence
   finsi
fincollatéral
dupliquer (δtransf.rang, duplication de (rang) ,
                         transfert de   (rang) )
      co produit le code dupliquant les valeurs des bornes sauvegardées
         par les opérandes de (rang) co
finséquence
```



décoration

déplasauve = déplacement de la sauvegarde des bornes dans la zone de transfert

information héritée

δtransf.bornes = descripteur représentant l'accès courant à la zone de transfert.

séquence
  descr δ
  collatéral
    séquence
      δ ← *modifieraccès (déplasauve de* (bornes) , δ*transf.bornes)*
          co descripteur représentant l'emplacement de sauvegarde de la borne co
      élaborer(unité-binf)
      sauveborne (δ, δ.unité-binf)
          co sauvegarde de la valeur de la borne inférieure dans la zone de transfert co
    finséquence
    séquence
      δ ← *modifieraccès (déplasauve de* (bornes) + *longsauveborne,*
                                                  δ*transf.bornes)*
          co descripteur représentant l'emplacement de sauvegarde de la la borne co
        élaborer (unité-bsup)
        sauveborne (δ, δ.unité-bsup)
          co sauvegarde de la valeur de la borne supérieure dans la zone de transfert co
    finséquence
  fincollatéral
finséquence

Remarque :
Les deux bornes sont toujours présentes sur l'arbre abstrait.

modèle sauveborne = (descr δdestination, δsource)

co δdestination : descripteur de la destination (zone de transfert)
δsource : descripteur de la valeur de la borne co

(:=,, δsource, δdestination)

finmodèle

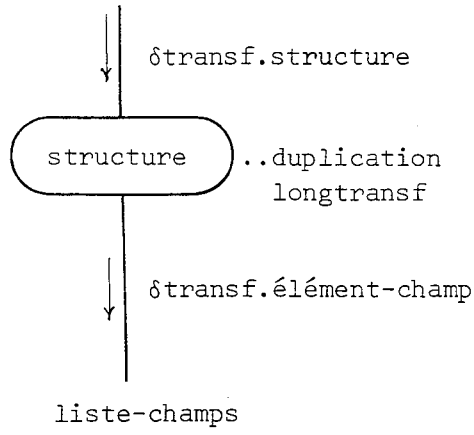
fonction longsauve = (ent nbdim) ent :

co calculer la longueur de la zone de transfert nécessaire à la sauvegarde des informations concernant un tableau à nbdim dimensions. Cette longueur est retournée comme résultat

co

finfonction

4.1.3.2 - (structure)



décorations

duplication = nombre de fois que le déclareur associé est "mis en facteur"

longtransf = longueur de la zone de transfert (associée à la structure) à dupliquer.

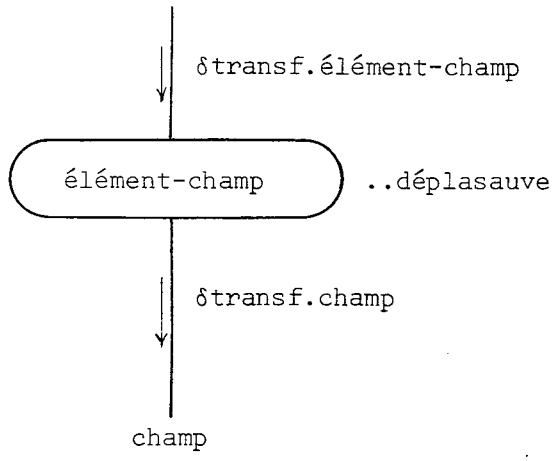
information héritée

δtransf.structure = descripteur représentant l'accès courant à la zone de transfert.

information à hériter

δtransf.élément-champ = descripteur représentant l'accès courant à la zone de transfert pour la sauvegarde des informations associées aux déclareurs des champs.

séquence
  pour tout arbre élément-champ ∈ liste-champs
  faire
    séquence
      δtransf.élément-champ ← δtransf.structure
      élaborer (élément-champ)
        co génération du code pour l'élaboration et la sauvegarde des bornes des tableaux apparaissant dans le champ co
    finséquence
  finfaire
    dupliquer (δtransf.structure, duplication de (structure) ,
                longtransf de (structure))
      co produit le code dupliquant la zone de transfert co
finséquence



décoration

déplasauve = déplacement de la sauvegarde des informations (associées au champ) dans la zone de transfert.

information héritée

δtransf.élément-champ = descripteur représentant l'accès courant à la zone de transfert.

information à hériter

δtransf.champ = descripteur représentant l'accès courant à la zone de transfert pour la sauvegarde des informations associées au déclareur du champ.

séquence

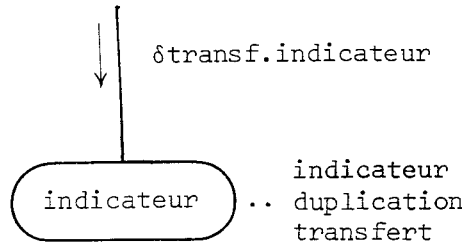
δtransf.champ ← modifieraccès (déplasauve de (élément-champ) ,
δtransf.élément-champ)

co modifier l'accès à la zone de transfert pour pouvoir sauvegarder les bornes apparaissant dans l'opérande champ co

élaborer (champ)

finséquence

4.1.3.3 - (indicateur)



décorations

indicateur = décoration représentant les caractéristiques de l'indicateur de mode

duplication = nombre de fois où l'indicateur est "mis en facteur"

transfert = longueur de la zone de transfert (associée au déclareur de l'indicateur) à dupliquer.

information héritée

δtransf.indicateur = descripteur représentant l'accès courant à la zone de transfert.

```
séquence
   descr δindic
      co descripteur pour représenter la routine associée à l'indicateur co
   δindic ← descrindic (indicateur de (indicateur) )
      co initialisation du descripteur de routine co
   routinedécla (δindic, δtransf.indication)
      co génération du code faisant l'appel à la routine associée à
         l'indicateur co
   dupliquer (δtransf.indication, duplication de (indicateur) ,
                                  transfert de (indicateur) )
      co génération du code dupliquant les valeurs sauvegardées par la
         routine associée à l'indicateur co
finséquence
```

4.1.4 - Fonctions utilisées par le traitement des déclareurs effectifs

En plus des fonctions globales, la description de ces traitements utilise les deux fonctions *modessrep* et *tailledyn* détaillées ci-après.

```
fonction modessrep = (type modeptr) type
    co
       délivrer le mode obtenu en dérepérant celui précisé par modeptr
    co
finfonction
```

```
fonction tailledyn = (descr δtransf) descr
    co
       δtransf : descripteur de la zone de transfert
       allouer et initialiser un descripteur (à partir de δtransf)
       pour qu'il représente l'accès à la taille dynamique sauvegardée,
       par l'action calcultaille, dans la zone de transfert
    co
finfonction
```

4.1.5 - Modèles utilisés par le traitement des déclareurs effectifs -

Les modèles spécifiques utilisés dans la description concernent les problèmes d'allocation d'espace.

Ils servent à la génération du code :

- appelant les routines d'allocation (bibliothèque) (*allocationlocale, allocationglobale*)
- modifiant les paramètres de ces routines (*incrtaillebiblio*)
- initialisant des bases avec les accès aux espaces de mémoire alloués par ces routines (*chargebases*).

modèle allocationlocale =
    co appel de la routine de bibliothèque allouant l'espace sur la zone dynamique de la pile co
  (appelbiblio, alloclocale, ,)
finmodèle

modèle allocationglobale =
    co appel de la routine de bibliothèque allouant l'espace sur le tas co
  (appelbiblio, allocglobale, ,)
finmodèle

modèle incrtaillebiblio = (descr δparam, δcte)
    co modification d'un paramètre d'une routine de bibliothèque
      δparam : descripteur représentant le paramètre à modifier
      δcte : descripteur de l'objet contenant la modification
    co
  (+:=, δparam, δcte,)
finmodèle

modèle chargebases = (descr δps, δpd, δaccès, δstat)

co δps et δpd sont les descripteurs pour les bases des parties statiques et dynamiques

δaccès : descripteur représentant la base de la zone allouée

δstat : descripteur de l'objet contenant la taille de la partie statique

code allouant et chargeant les bases pour accèder à la partie statique et la partie dynamique

co

(allouer, base, δaccès, δps)

co allocation, initialisation de la base de la partie statique.

δps représente la ressource servant de base co

(allouer, base, δaccès, δpd)

(+:=, δpd, δstat,)

co allocation, initialisation de la base de la partie dynamique

δpd représente la ressource servant de base co

finmodèle

4.1.6 - Allocation d'espace pour les générateurs à parties dynamiques -

La zone de transfert, une fois initialisée par la routine appelée, va être utilisée pour calculer la taille à allouer.
Le stade suivant est donc la génération de code pour cette utilisation. Cela concerne les traitements des noeuds :

- (déclaration-d-variable) , (déclaration-i-variable) dans lesquels nous utilisons l'action *allocdynamiquevar*
- (d-générateur) , (i-générateur) dans lesquels nous utilisons l'action *allocdynamiquegén.*

L'espace alloué doit ensuite être structuré pour contenir la représentation d'une valeur du mode demandé. C'est l'action *partage* qui se charge de la génération de code correspondante.

```
action allocdynamiquegen = (descr δtransf, type mode, localité local)
    co génération de code pour appeler la routine d'allocation d'espace
       dans la pile ou le tas,et structurer cet espace en fonction du
       mode de la valeur qu'il doit contenir.
       δtransf : descripteur de la zone de transfert contenant notamment
                 la taille de la partie dynamique de l'objet
       local   : allocation à produire dans la pile ou dans le tas co
  descr δparstat, δpardyn
    co descripteurs pour les bases des parties statiques et dynamiques co
  taillegenetvartas (δtransf, mode)
    co génération du code pour le calcul de la taille à allouer.
       Cette taille est mise dans le paramètre bibliothèque représenté
       par parambibliothèque co
  si local
  alors allocationlocale
  sinon allocationglobale
  finsi
    co génération du code appelant la routine d'allocation dans la pile
       ou dans le tas. Le résultat de l'allocation (c'est-à-dire, la base
       de l'espace alloué) est représenté par le descripteur
        résultbibliothèque co
  basestatdyn (δparstat, δpardyn, mode)
    co génération de code pour charger en ressource-machine, la base de
       la partie statique et celle de la partie dynamique de l'espace
       alloué co
  partage (δparstat, δpardyn, δtransf, mode)
    co génération de code pour partager et structurer l'espace alloué en
       fonction du mode de la valeur qu'il doit contenir co
finaction
```

action allocdynamiquevar = (descr δidf, δtransf, type mode, localité local)
co génération de code pour appeler la routine d'allocation d'espace dans la pile ou le tas, et structurer cet espace en fonction du mode de la valeur qu'il doit contenir.
δidf = descripteur représentant l'objet associé à l'identificateur déclaré
δtransf = descripteur de la zone de transfert contenant notamment la taille de la partie dynamique de l'objet associé à l'identificateur
local = allocation à produire dans la pile ou dans le tas co
descr δparstat, δpardyn
co descripteurs pour les bases des parties statiques et dynamiques co
si local
alors
allocationlocale
co génération du code appelant la routine d'allocation dans la pile. La base de l'espace alloué est représentée par le descripteur δrésultbibliothèque co
allocbase (δpardyn, δrésultbibliothèque)
co génération de code chargeant, en ressource-machine, la base de la partie dynamique de l'espace alloué co
partage (δidf, δpardyn, δtransf, mode)
co génération de code pour partager et structurer l'espace alloué en fonction du mode de la valeur qu'il doit contenir co
co l'accès à la partie statique de l'objet associé à l'identificateur est représenté dans ce cas par δidf co
sinon
taillegenetvartas (δtransf, mode)
co génération du code pour le calcul de taille à allouer (partie statique + partie dynamique). Cette taille est mise dans le paramètre bibliothèque représenté par δparambibliothèque co
allocationglobale
co génération du code appelant la routine d'allocation dans le tas. Le résultat de l'allocation (c'est-à-dire, la base de l'espace alloué) est représenté par le descripteur δrésultbibliothèque co
basestatdyn (δparstat, δpardyn, mode)
co génération de code pour charger, en ressource-machine, la base de la partie statique et celle de la partie dynamique de l'espace alloué co

```
        partage (δparstat, δpardyn, δtransf, mode)
            co génération de code pour partager et structurer l'espace
               alloué en fonction du mode de la valeur qu'il doit contenir co
        copieresult (δidf)
            co génération de code pour établir la relation "posséder" co
    finsi
finaction

action taillegenetvartas = (descr δtransf, type mode)
        co δtransf : descripteur de la zone de transfert contenant notamment la
                     taille de la partie dynamique
             mode  : mode de la valeur co
    descr δconstante
        co descripteur pour la taille statique co
    δconstante ← constante (taille (mode))
        co δconstante représente la taille statique du mode co
    chargeparamètre (δparambibliothèque, tailledyn (δtransf)) ;
        co génération du code chargeant la taille dynamique dans la zone des
           paramètres (δparambibliothèque) co
    incrtaillebiblio (δconstante)
        co génération du code ajoutant au paramètre (représenté par
           δparambibliothèque), la taille de la partie statique co
finaction

action basestatdyn = (descr δps, δpd, type mode)
        co δps et δpd sont les descripteurs qui vont représenter la base de la
           partie statique et celle de la partie dynamique de l'espace alloué
           δrésultbibliothèque représente la base de l'espace alloué co
    descr δconstante
        co descripteur pour la taille statique co
    δconstante ← constante (taille (mode))
        co  constante représente la taille statique du mode co
    chargebases (δps, δpd, δrésultbibliothèque, δconstante)
        co alloue des ressources et les initialise avec les bases des parties
           statiques et dynamiques. Les descripteurs δps et δpd représentent
           ces deux bases co
finaction
```

4.2 - Traitement des noeuds associés aux générateurs et aux déclarations -

Ces noeuds, qui correspondent à six constructions du langage, sont :

- déclaration de mode

  (déclaration-d-mode) , (déclaration-i-mode)

- générateur

  (s-générateur) , (d-générateur) , (i-générateur)

- déclaration de variable

  (déclaration-s-variable), (déclaration-d-variable), (déclaration-i-variable)

- déclaration d'identité

  (déclaration-identité)

- déclaration d'opérateur

  (déclaration-opérateur)

- déclaration d'étiquette

  (étiquette)

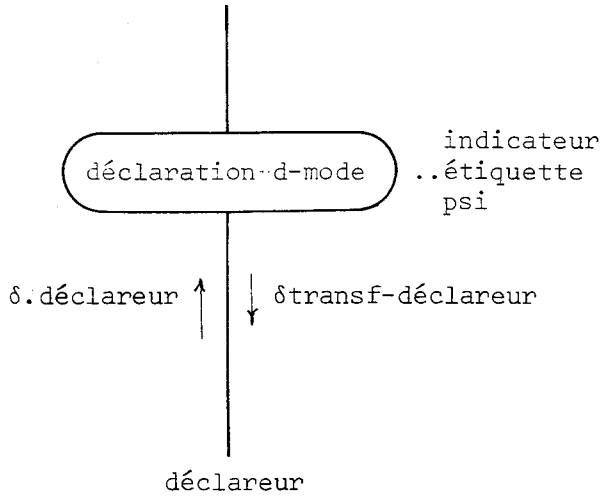
4.2.1 - Déclarations de mode

4.2.1.1 - (déclaration-d-mode)

Le déclareur associé à cette déclaration a des parties dynamiques et n'est pas réduit à un simple indicateur.

Exemples :
```
mode m1 = struct ([a:10, 8] ent i, j, réel k) ;
mode m2 = struct (m1 x, bool b) ;
```

Ce noeud a comme opérande l'arbre de son déclareur dont le parcours va permettre la génération d'une routine. Cette routine est à associer à l'indicateur déclaré.



décorations

indicateur = décoration représentant les caractéristiques statiques de l'indicateur déclaré

étiquette = étiquette représentant le point d'entrée de la routine associée au déclareur

psi = taille de la zone des variables de la routine associée au déclareur

information à hériter

δtransf.déclareur = descripteur représentant l'accès à la zone de transfert associée au déclareur. Ce descripteur est construit par l'action *élaborerdéclareur*.

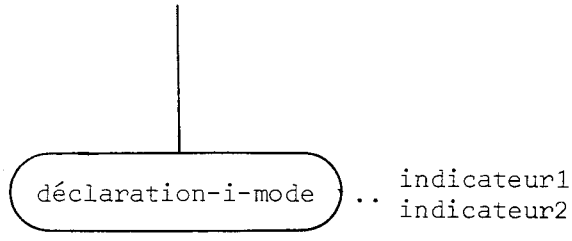
séquence
descr δind
co allocation d'un descripteur représentant l'objet associé à l'indicateur co
δind ← *descrindic (indicateur de* (déclaration-d-mode))
co δind *décrit l'objet, à l'exécution, associé à l'indicateur co*
élaborerdéclareur (déclareur, étiquette de (déclaration-d-mode) ,
psi de (déclaration-d-mode))
co génération de la routine évaluant et sauvegardant les bornes du déclareur dans la zone de transfert. Synthèse d'un descripteur de routine : δ.*déclareur co*
*affectation (*δind, δ.*déclareur, routine)*
finséquence

4.2.1.2 - (déclaration-i-mode)

Le déclareur associé à cette déclaration est réduit à un indicateur dont le déclareur correspondant a des parties dynamiques

Exemple : *mode m1 = m2 ;*
avec *mode m2 = [i:j] [1:10] réel ;*

Une routine est déjà produite pour l'indicateur formant le déclareur. Il suffit d'associer la même routine au nouvel indicateur.



décorations

indicateur1 = descripteur représentant les caractéristiques de l'indicateur déclaré (*m1*)

indicateur2 = descripteur représentant les caractéristiques de l'indicateur utilisé (*m2*)

séquence
descr δsource, δdestination
δsource ← *descrindic (indicateur2 de* (*déclaration i-mode*))
δdestination ← *descrindic (indicateur1 de* (*déclaration i-mode*))
*affectation (*δdestination, δsource, *statique)*
co génération du code permettant d'affecter des objets de mode
proc co
finséquence

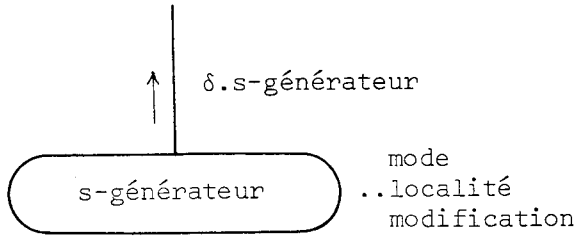
4.2.2 - Générateurs

4.2.2.1 - (s-générateur)

Le déclareur associé n'a pas de parties dynamiques. La taille à allouer (dans la zone dynamique ou dans le tas) est donc connue.

Exemple :

```
début
   rep réel x,y ;
   x:= tas réel ; y:= loc réel ;
   ...
   fin
```



décorations

mode = mode du générateur

localité = décoration indiquant si l'allocation est à faire dans la pile (*loc*) ou dans le tas (*tas*)

modification = liste des modifications à appliquer à la valeur fournie par le générateur

```
séquence
   descr δconstante, δres
   δconstante ← constante (taille (modessrep (mode de (s-générateur) )))
        co range la taille dans la table des constantes et initialise le
           descripteur co
   chargeparamètre (δparambibliothèque, δconstante)
        co génération du code chargeant le paramètre de la routine
           bibliothèque d'allocation d'espace co
   si localité de (s-générateur)

        alors
                allocationlocale
      . sinon
                allocationglobale
   finsi
        co génération du code appelant la routine d'allocation d'espace dans
           la pile ou le tas co
   δres ← construires-générateur
        co construction du descripteur du générateur co
   gmodif (δres, modification de (s-générateur) )
   δ.s-générateur ← δres
finséquence


fonction construires-générateur = descr
   descr δ
     co initialiser le descripteur δ pour qu'il décrive le générateur co
   δ ← δrésultbibliothèque
     co initialise δ pour qu'il décrive la ressource contenant le résultat
        de l'allocation (c'est-à-dire la base de l'espace alloué) du
        générateur. co
   complétermode (mode de (s-générateur) )
     co δ décrit un générateur du mode demandé co
finfonction
```

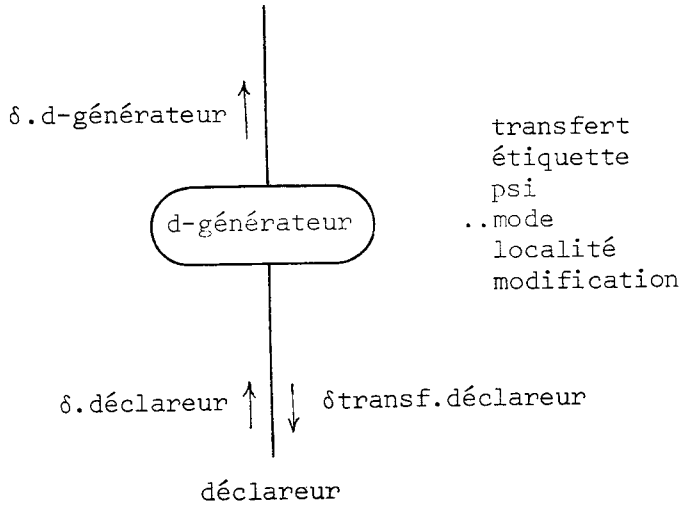
4.2.2.2 - (d-générateur)

Le déclareur associé a des parties dynamiques et n'est pas réduit à un indicateur.
Le traitement consiste à produire :

- la routine, en parcourant l'arbre du déclareur (opérande du noeud)
- l'appel de cette routine
- le calcul de la taille à allouer
- l'allocation et la structuration de l'espace alloué.

Exemple :

```
début
   rep [,] ent rrte ;
   ent a,b ;
   ...
   rrte := loc [1:a, 1:b] ent ;
   ...
fin
```



décorations

transfert = taille de la zone de transfert associée au déclareur du générateur

étiquette = étiquette représentant le point d'entrée de la routine associée au déclareur

psi = taille de la zone des variables de la routine

mode = mode du générateur

localité = décoration indiquant si l'allocation est à faire dans la pile ou dans le tas

modification = liste des modifications à appliquer à la valeur fournie par le générateur.

information à hériter

δtransf.déclareur = descripteur représentant l'accès à la zone de transfert associée au déclareur. Ce descripteur est construit par l'action *élaborerdéclareur*.

séquence

descr δ*sauvegarde*, δ*res*

co descripteurs pour la zone de transfert co

élaborerdéclareur (déclareur, étiquette de (*d-générateur*) ,

psi de (*d-générateur*))

co génération de la routine évaluant et sauvegardant les bornes dans la zone de transfert. Synthèse d'un descripteur de routine : δ.*déclareur co*

δ*sauvegarde* ← *allocpse (transfert de* (*d-générateur*))

co allocation dans la zone d'évaluation de la zone de transfert et initialisation de la partie "accès" du descripteur δ*sauvegarde co*

*routinedécla (*δ.*déclareur,* δ*sauvegarde)*

co génération du code pour l'appel de la routine associée au déclareur du générateur co

*calcultaille (*δ*sauvegarde, mode de* (*d-générateur*))

co génération de code permettant à partir des informations rangées dans la zone de transfert, et du mode, de calculer la taille dynamique de l'objet. Cette taille est rangée dans la zone de transfert co

allocdynamiquegén (δsauvegarde, modessrep (mode de (d-générateur)),
localité de (d-générateur))

co génération du code appelant la routine d'allocation d'espace sur la pile ou le tas co

δres ← construire d-générateur

gmodif (δres, modification de (d-générateur))

δ.d-générateur ← δres

finséquence

fonction construired-générateur = descr

descr δ

co initialiser le descripteur δ pour qu'il représente le générateur co

δ ← δrésultbibliothèque

complétermode (δ, mode de (d-générateur))

finfonction

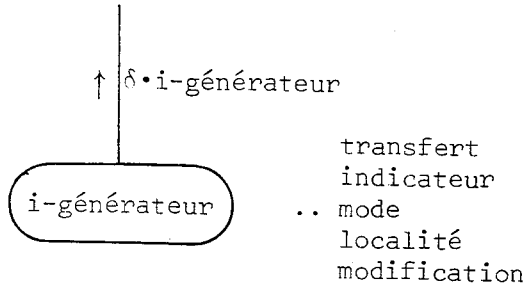
4.2.2.3 - (i-générateur)

Le déclareur associé est réduit à un indicateur dont le déclareur correspondant a des parties dynamiques.
La routine à appeler est celle associée au déclareur de l'indicateur.
Le traitement consiste donc à produire :

- l'appel à cette routine
- le calcul de la taille à allouer
- l'allocation et la structuration de l'espace alloué.

Exemple :

```
début
   ent a,b ;
   ...
   mode m1 = [a:b] réel ;
   rep [ ]réel rrtr ;
   rrtr := loc m1 ;
   ...
fin
```



décorations

transfert	= liste de la zone de transfert associée au déclareur de l'indicateur utilisé
indicateur	= décoration représentant les caractéristiques de l'indicateur utilisé
mode	= mode du générateur
localité	= décoration indiquant si l'allocation est à faire dans la pile ou dans le tas
modification	= liste des modifications à appliquer à la valeur fournie par le générateur

séquence

descr δsauvegarde, δind, δres

co descripteurs pour la zone de transfert et l'indicateur co

δsauvegarde ← *allocpse (transfert de* (*i-générateur*))

co allocation dans la zone d'évaluation de la zone de transfert et initialisation de la partie "accès" du descripteur δsauvegarde *co*

δind ← *descrindic (indicateur de* (*i-générateur*))

co descripteur représentant la routine associée à l'indicateur co

*routinedécla (*δind, δsauvegarde*)*

co génération du code faisant l'appel à la routine du générateur co

allocdynamiquegén (sauvegarde, modessrep (mode de (*i-générateur*)),

localité de (*i-générateur*))

co génération du code appelant la routine d'allocation d'espace dans la pile ou le tas co

δres ← *construirei-générateur*

*gmodif (*δres, *modification de* (*i-générateur*))

δ.*i-générateur* ← δres

finséquence

```
fonction construire i-générateur = descr
   descr δ
     co initialiser δ pour qu'il représente le générateur co
   δ ← δrésultbibliothèque
   complétermode (δ, mode de (i-générateur) )
finfonction
```

4.2.3 - Déclarations de variable et déclaration d'identité -

Ces quatre noeuds ont un fils représentant une liste.
L'existence de cette liste est due aux possibilités (définies dans le langage) de "mise en facteur" du déclareur dans les déclarations de variable et d'identité.

Exemples :

```
[1:10, a:b] bool x,y ;
ent a := 1, b,c := 3 ;
rep rep réel t = loc rep réel, u = ... ;
```

Chaque noeud (définition-idf) de la liste correspond à un identificateur déclaré. Ils ont tous un opérande correspondant :

- à la partie droite de la déclaration, dans le cas d'une déclaration d'identité
- à la valeur servant à linitialiser la variable (valeur *fant* si l'utilisateur ne l'a pas précisée) dans le cas d'une déclaration de variable.

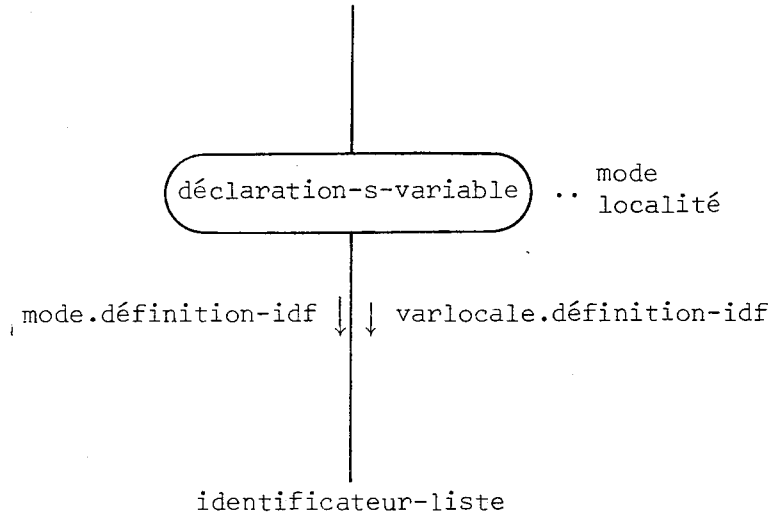
4.2.3.1 - Déclarations de variable -

4.2.3.1.1 - (déclaration-s-variable)

Le déclareur n'a pas de parties dynamiques

Exemples : `ent x ;  rep [ ] bool t := ... ;`

Les espaces, pour les représentations des valeurs possédées par les identificateurs de la liste opérande du noeud, sont réservées dans la zone des variables.



décorations

mode = mode de l'identificateur

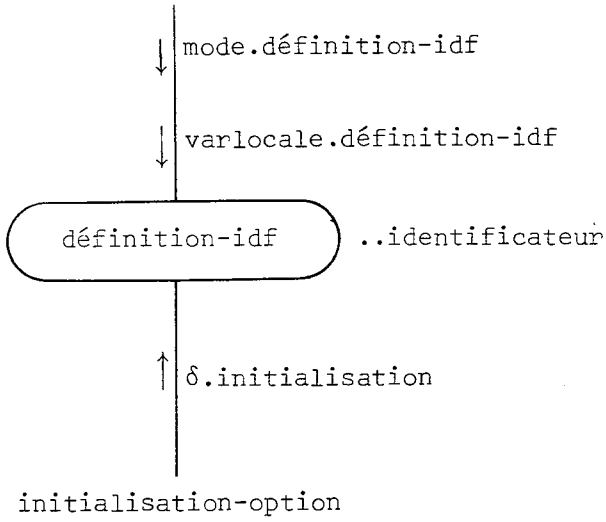
localité = décoration indiquant si l'allocation est à faire dans la pile ou dans le tas.

informations à hériter

mode.définition-idf = mode du déclareur après dérepérage

varlocale.définition-idf = indique si l'allocation est à faire dans la pile ou le tas.

pour tout arbre définition-idf ∈ identificateur-liste
faire
séquence
mode.définition-idf ← modessrep (mode de déclaration-s-variable)
varlocale.définition-idf ← localité de déclaration-s-variable
co transmission, sur une définition de variable de la liste, de la localité et du mode du générateur co
élaborer (définition-idf)
co génération du code pour l'élaboration d'un élément de la liste de déclarations co
finséquence
finfaire



décoration

identificateur = décoration représentant les caractéristiques de l'identificateur déclaré.

informations héritées

mode.définition-idf = mode du déclareur dérepéré, c'est-à-dire, mode de la valeur que doit fournir le fils initialisation (s'il existe)

varlocale.définition-idf = indique si l'allocation est à faire dans la pile ou le tas.

séquence

descr δ, δ1
δ ← *descridf (identificateur de définition-idf*
co δ *est le descripteur représentant l'objet associé à l'identificateur co*
si ¬ *varlocale.définition-idf*
alors
δ1 ← *constante (taille (mode.définition-idf))*
chargeparamètre (&parambibliothèque, δ1)
co génération du code pour chargement du paramètre de la routine d'allocation d'espace co
allocationglobale
co génération de code pour appel de la routine d'allocation co
copierésult (δ)
co génération du code réalisant la relation "posséder" co
finsi

élaborer (initialisation)

co génération de code pour l'élaboration de la partie initialisation co

affectation (δ, δ.initialisation, statique)

finséquence

4.2.3.1.2 - (déclaration-d-variable)

Le déclareur associé a des parties dynamiques et n'est pas réduit à un indicateur.

Le traitement consiste à produire :

- la routine, en parcourant l'arbre du déclareur (premier opérande)
- l'appel de cette routine
- le calcul de la taille à allouer
- l'allocation et la structuration de l'espace alloué (dans la zone des variables, dans la zone dynamique et/ou dans le tas) pour chaque variable de la liste d'identificateurs (deuxième opérande).

Exemple :

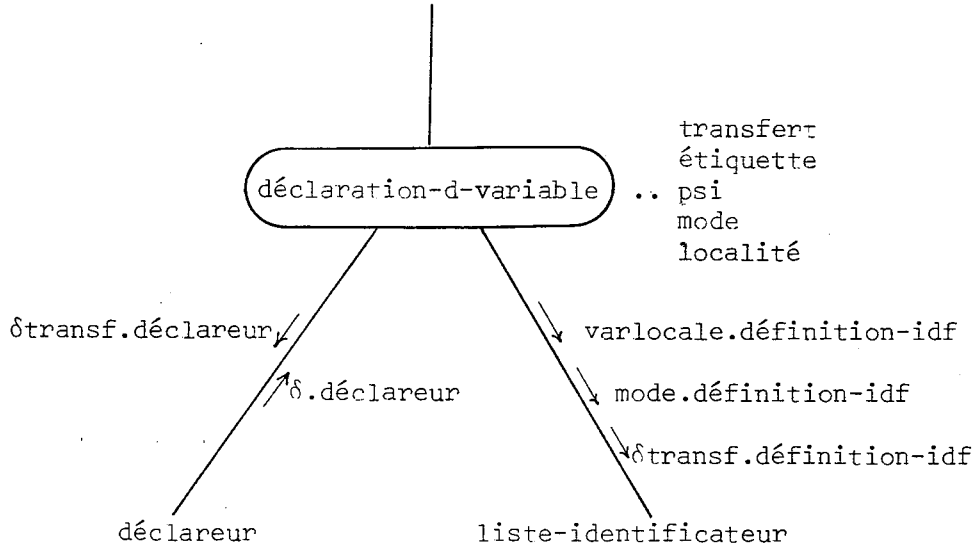
début

ent a,b ;

*[1:a*b+7] struct (réel c, compl d) ;*

⋮

fin



décorations

transfert = taille de la zone de transfert associée au déclareur de la déclaration

étiquette = étiquette représentant le point d'entrée de la routine associée au déclareur

psi = taille de la zone des variables de la routine

mode = mode de l'identificateur

localité = indicateur d'allocation dans la pile ou dans le tas.

informations à hériter

δtransf.déclareur = descripteur représentant l'accès à la zone de transfert associée au déclareur. Ce descripteur est construit par l'action *élaborerdéclareur*

varlocale.définition-idf = indique si l'allocation est à faire dans la pile ou dans le tas

mode.définition-idf = mode du déclareur après dérepérage

δtransf.définition-idf = descripteur décrivant la zone de transfert associée au déclareur.

séquence

descr δtransfert

élaborerdéclareur (déclareur, étiquette de (*déclaration d-variable*) ,
psi de (*déclaration d-variable*))

co génération de la routine évaluant et sauvegardant les bornes dans la zone de transfert co

δtransfert ← allocpse (transfert de (*déclaration d-variable*))

co allocation dans la zone d'évaluation de la zone de transfert et initialisation des informations d'"accès" du descripteur δtransfert co

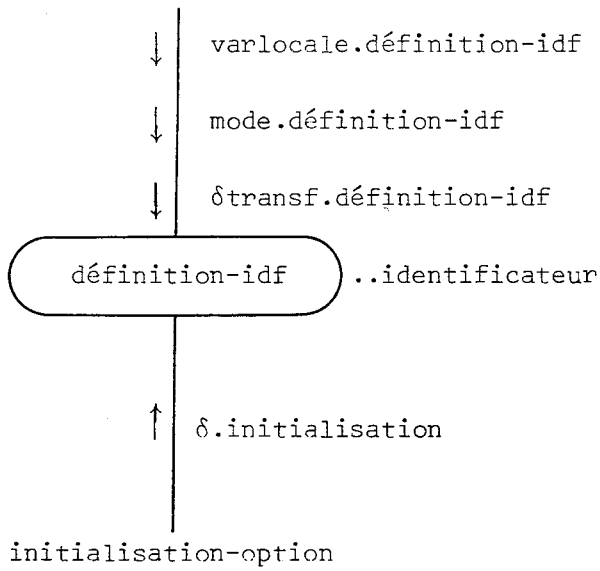
routinedécla (δ.déclareur, δtransfert)

co génération du code pour l'appel de la routine associée au déclareur co

calcultaille (δtransfert, modessrep (mode de (*déclaration d-variable*)))

co génération du code permettant à partir des informations rangées dans la zone de transfert et du mode, de calculer la taille dynamique de l'objet. Cette taille est rangée dans la zone de transfert co

```
pour tout arbre définition-idf ∈ liste-identificateur
faire
   séquence
      varlocale.définition-idf ← localité de (déclaration-d-variable)
      mode.définition-idf ← modessrep (mode de (déclaration-d-variable) )
      δtransf.définition-idf ← δtransfert
           co informations héritées sur chaque élément de la liste co
      élaborer (définition-idf)
           co génération du code pour élaborer un élément de la liste
              de déclarations co
   finséquence
finfaire
finséquence
```



décoration

identificateur = décoration représentant les caractéristiques de l'identificateur déclaré.

informations héritées

mode.définition-idf = mode du déclareur dérepéré, c'est-à-dire mode de la valeur que doit fournir le fils initialisation (s'il existe)

varlocale.définition-idf = indique si l'allocation est à faire dans la pile ou le tas

δtransf.définition-idf = descripteur décrivant la zone de transfert associée au déclareur.

séquence

descr δ

δ ← *descridf (identificateur de* (définition-idf))

co δ *est le descripteur représentant l'objet associé à l'identificateur co*

allocdynamiquevar (δ, δtransf.définition-idf, mode.définition-idf, varlocale.définition-idf)

co génération du code pour allouer un emplacement dans le tas ou la pile, et le structurer en fonction du mode de la valeur qu'il doit contenir. La partie statique de l'objet représenté par δ est initialisée co

élaborer (initialisation)

co génération du code pour l'élaboration de la valeur d'initialisation co

affectation (δ, δ.initialisation, dynamique)

co génération de code pour l'affectation de la valeur d'initialisation co

finséquence

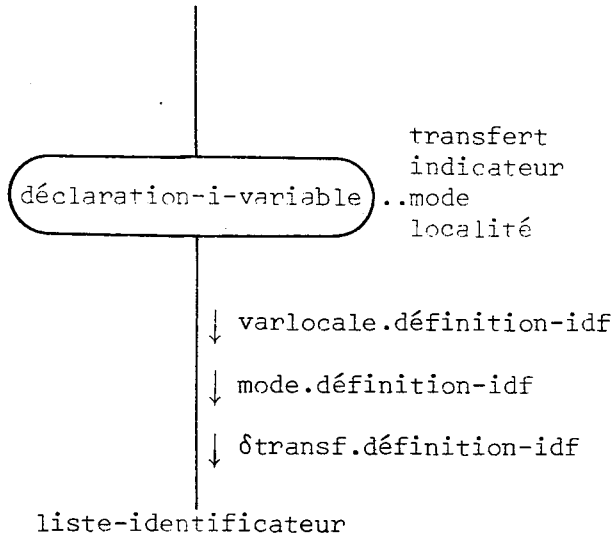
4.2.3.1.3 - (déclaration-i-variable)

Le déclareur associé est réduit à un indicateur dont le déclareur correspondant a des parties dynamiques.
La routine à appeler est celle associée au déclareur de l'indicateur.
Le traitement consiste à produire :

- l'appel à cette routine
- le calcul de la taille à allouer
- l'allocation et la structuration de l'espace alloué (dans la zone des variables, dans la zone dynamique et/ou dans le tas) pour chaque variable de la liste d'identificateurs (opérande).

Exemple :

```
début
   ent a,b ;
   ...
   mode m1 = [a:b*2] réel ;
   m1 tr ;
   ...
fin
```



décorations

transfert = taille de la zone de transfert associée au déclareur de la déclaration

indicateur = décoration représentant les caractéristiques de l'indicateur utilisé

mode = mode de l'identificateur

localité = indicateur d'allocation dans la pile ou dans le tas.

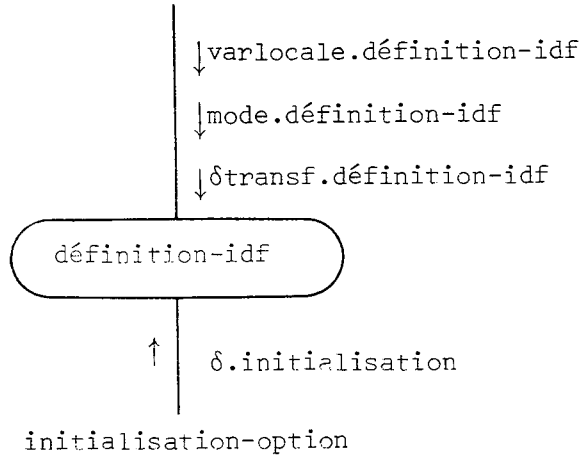
informations à hériter

varlocale.définition-idf = indique si l'allocation est à faire dans la pile ou dans le tas

mode.définition-idf = mode du déclareur après dérepérage

δtransf.définition-idf = descripteur décrivant la zone de transfert associée au déclareur.

```
séquence
    descr δtransfert, δind
    δtransfert ← allocpse (transfert de (déclaration i-variable) )
        co allocation dans la zone d'évaluation de la zone de transfert et
           initialisation des informations d'"accès" du descripteur
           δtransfert co
    δind ← δdescrindic (indicateur de (déclaration i-variable) )
        co descripteur représentant la routine associée à l'indicateur co
    routinedécla (δind, δtransfert)
        co génération du code pour l'appel de la routine associée à
           l'indicateur co
    calcultaille (δtransfert, modessrep (mode de (déclaration i-variable) )
        co génération du code permettant à partir des informations rangées
           dans la zone de transfert et du mode, de calculer la taille
           dynamique de l'objet. Cette taille est rangée dans la zone de
           transfert co
    pour tout arbre définition-idf ∈ liste-identificateurs
    faire
        séquence
            varlocale.définition-idf ← localité de (déclaration-i-variable)
            mode.définition-idf ← modessrep (mode de (déclaration-i-variable) )
            δtransf.définition-idf ← δtransfert
                co informations héritées sur chaque élément de la liste co
            élaborer (définition-idf)
                co génération du code pour élaborer un élément de la liste
                   de déclarations co
        finséquence
    finfaire
finséquence
```



Le traitement est identique à celui du noeud (définition-idf) dans le cas des déclarations nécessitant un noeud (déclaration-d-variable)

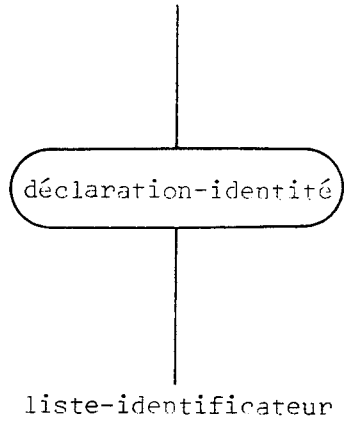
4.2.3.2 - Déclaration d'identité -

(déclaration-identité)

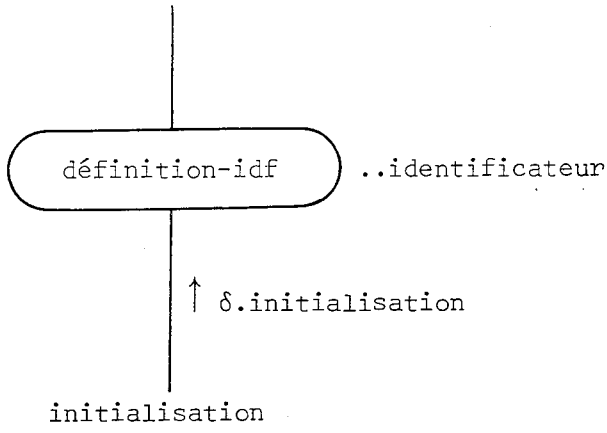
Le traitement consiste à produire, pour chaque identificateur de la liste, l'évaluation de la partie droite de la déclaration et l'initialisation de l'espace (avec allocation en pile dynamique si le déclareur commun possède des parties dynamiques) associé.

Exemple :

```
début
   [ , ] ent a = ... ;
   réel x = ... ;
   mode m1 = rep compl ;
   m1 z = ... ;
   ...
fin
```




```
pour tout arbre définition-idf ∈ liste-identificateur
faire
   élaborer (définition-idf)
finfaire
```



décoration

identificateur = décoration représentant les caractéristiques de l'identificateur déclaré.

séquence

descr δ

co descripteur pour l'objet associé à l'identificateur co

δ ← *descridf (identificateur de* (définition-idf))

élaborer (initialisation)

co génération du code évaluant la partie droite de la déclaration d'identité co

affectation (δ, δ.initialisation, identité)

co génération du code initialisant l'objet représenté par l'identificateur après lui avoir alloué et structuré de l'espace, s'il possède des parties dynamiques co

finséquence

4.2.4 - Déclaration d'opérateur -

(déclaration-opérateur)

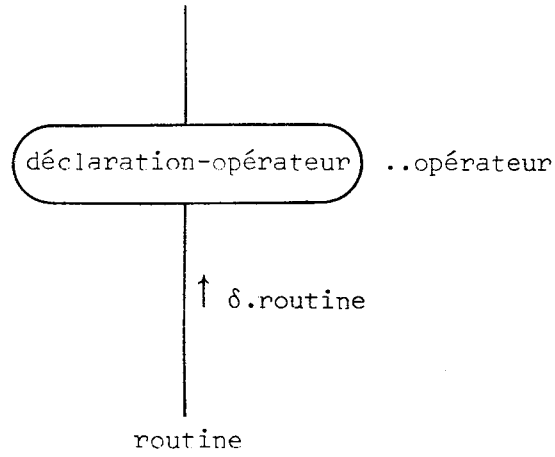
L'opérande de ce noeud est une routine. Le traitement consiste donc à produire le code de cette routine et à établir la relation "posséder" entre l'opérateur et la routine.

Rappelons que la représentation d'une routine est formée de deux pointeurs :

- l'un vers le code engendré pour la routine
- l'autre vers la base de l'environnement de définition de la routine.

Exemple :

```
début
   op plus2fois = (ent a,b) ent : a+b+b ;
   op plus3fois = (ent a,b) ent : a plus2fois b+b ;
   ...
fin
```



décoration

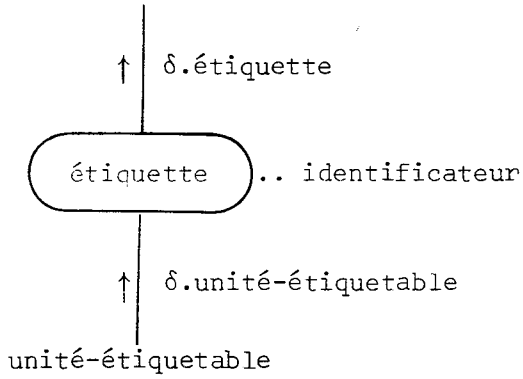
opérateur = décoration représentant les caractéristiques de l'opérateur déclaré.

```
séquence
   descr δ
        co descripteur pour l'objet associé à l'opérateur co
   δ ← descrop (opérateur de (déclaration-opérateur) )
   élaborer (routine)
        co génération de code et synthèse d'un descripteur de routine co
   affectation (δ, δ.routine, statique)
finséquence
```

4.2.5 - Déclaration d'étiquette -

(étiquette)

Le traitement consiste à engendrer l'étiquette programme puis le code correspondant à l'unité étiquetée.



décoration

identificateur = décoration représentant les caractéristiques de l'étiquette.

séquence

étiquette étiqprog

co déclaration pour l'étiquette du programme

étiqprog ← étiq (identificateur de (étiquette))

co construction d'une décoration étiquette co

génétiq (étiqprog)

co génération de l'étiquette du programme co

élaborer (unité-étiquetable)

δ.étiquette ← δ.unité-étiquetable

finséquence

```
modèle génétiq = (étiquette prétiq)
     co génération d'une étiquette du programme
        prétiq : étiquette à générer co
   (défétiq, prétiq,,)
finmodèle
```

```
fonction étiq  = (identificateur idf) étiquette
     co
        à partir d'une décoration de type identificateur associée à une
        déclaration d'étiquette, délivrer une décoration de type étiquette.
     co
finfonction
```

4.3 - Conclusion -

Nous venons d'étudier et de décrire le traitement réalisé pour les déclarations et les générateurs en Algol 68. Rappelons que les demandes d'espace (de taille statiquement connue ou non) se font au moyen d'un générateur explicite ou sous-entendu (cas des déclarations de variables).

Les déclarations de mode et d'opérateur présentent des caractéristiques peu classiques et ont nécessité l'emploi de nouvelles techniques de compilation.

5 - PRODUCTION DE VALEURS -

Indépendamment des modifications qui permettent d'obtenir une valeur à partir d'une autre valeur, il existe des constructions du langage :

- délivrant une valeur composée (tableau ou structure) à partir de valeurs indépendantes (collatéral non neutre)

Exemple :

```
début
   ent a,b ;
   ...
   [1:3] ent t = (a,1,b);
   struct (ent x,y,z) ste ;
   ste := (a,1,b);
   ...
fin
```

- délivrant une valeur à partir d'une valeur composée (sélection simple, indexation)

Exemple :

```
début
   ent a,b ;
   ...
   [1:a, 1:b] ent tt ;
   struct (ent x, réel y) ster ;
   ...
   c := tt [a-1, b-1] ;
   d := x de ster ;
   ...
fin
```

- délivrant, à partir d'une valeur composée de type tableau, une sous-valeur, elle-même de type tableau (sélection multiple, tranche)

Exemple :

```
début
   [1:3] ent t,u ;
   ent a,b ;
   ...
   [1:a, 1:b] ent tt ;
   [1:3] struct (ent x, réel y) ster ;
   ...
   t := tt [1:3, b] ;
   u := x de ster ;
   ...
fin
```

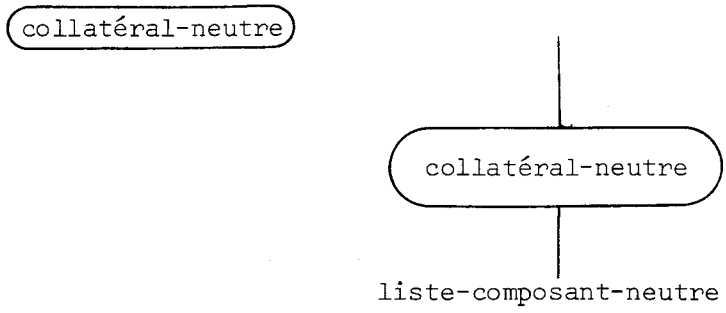
Ces constructions nécessitent généralement (sauf en ce qui concerne la sélection simple) une génération abondante de code car elles manipulent des valeurs ayant des parties dynamiques.

5.1 - (collatéral)

5.1.1 - Cette construction peut ne pas retourner de valeur (plus précisément, peut retourner une valeur neutre). Dans ce cas le traitement se résume à produire le code pour chaque élément du collatéral dans un ordre quelconque.

Exemple :

```
début
   ent a ; lire (a) ;
   réel b := ... ;
   proc p = bool : ... ;
   ... ;
   (a +:= 1, b := b/2, p) ;
   ...
fin
```



pour tout arbre composant ∈ liste-composant-neutre
faire
élaborer (composant)
finfaire

5.1.2 - Si, par contre, cette construction délivre un tableau ou une structure, il faut construire la représentation de la valeur correspondante. Cette valeur étant formée à partir de valeurs indépendantes, le code produit doit en "fabriquer" complètement la représentation.

Le noeud a comme fils l'ensemble des arbres correspondant aux éléments du collatéral. Cet ensemble est représenté sous une forme explicite de liste. Bien que les éléments s'évaluent collatéralement, la valeur finalement construite doit tenir compte de l'ordre dans lequel le programme les spécifie. L'utilisation d'une représentation explicite de liste nous permet de synthétiser, en fin de traitement de la liste, un ensemble formé des descripteurs des éléments, l'ordre de ces descripteurs étant celui précisé dans le programme.

Une fois terminée la génération de code pour les éléments, le traitement diffère suivant le mode de la valeur à obtenir :

- s'il s'agit d'une structure, seule sa représentation statique est construite. Le code produit réalise, dans un ordre quelconque la mise en place de la représentation statique de la valeur de chaque champ (par copie de la partie statique de l'élément correspondant du collatéral) ;
- s'il s'agit d'un tableau :
    - la représentation statique est construite dans la zone d'évaluation,
    - de l'espace est alloué pour la représentation de sa partie dynamique

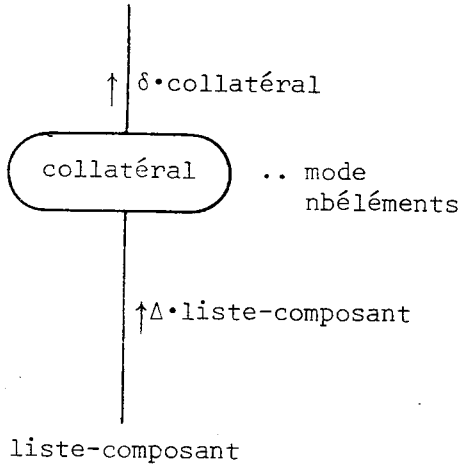
- l'initialisation de sa représentation statique étant faite, ses éléments sont mis en place un par un (dans un ordre quelconque), en vérifiant chaque fois leur cohérence mutuelle.

Exemple :

```
début
   [1:3, 1:2] ent tt ;
   [1:2] ent t1, t2 ;
   [2:5] ent terr ;
   ...
   tt := (t1, (3,4), t2) ;
   ...
fin
```

Le collatéral de tableau $(t_1, (3,4), t_2)$ est correct car les trois éléments t_1, $(3,4)$, t_2 sont tous des tableaux de mode [1:2] ent. Par contre $(t_1, terr, t_2)$ ne serait pas réalisable car le deuxième élément, de mode [2:5] ent ,ne serait pas compatible avec les deux autres.

collatéral



décorations

mode : mode de la construction collatérale.

nbéléments : nombre d'éléments de la phrase collatérale.

```
séquence
    descr δpartiestatique
    élaborer (liste-composant)
      co génération de code pour l'élaboration de la liste d'unité co
    δpartiestatique ← allocpse (taille (mode de (collatéral) ))
      co allocation de la partie statique de l'objet co
    δpartiestatique ← complétermode (δpartie statique, mode de (collatéral) )
    géncollatéral (Δ.liste-composants, δpartiestatique)
      co génération du code pour la construction d'un objet de mode structure
         ou tableau co
    δ.collatéral ← δpartiestatique
finséquence

action géncollatéral = (ensemble descr Δ, descr δ)
     co Δ : ensemble des descripteurs représentant les valeurs des composants
            de la phrase collatérale
        δ : descripteur représentant la partie statique de l'objet co
   cas type modedescr (δ) dans
      structure :
           co l'objet à construire est une structure co
        descr δchamp
        pour tout descr δk ∈ Δ
        faire
           δchamp ← sélectionchamp (δ, k, modedescr (δ))
                co met à jour le descripteur δchamp représentant la partie
                   statique du k^ième champ à partir du mode de la structure co
           affectation (δchamp, δk, statique)
                co génération du code recopiant la partie statique du champ co
        finfaire
          co fin du cas structure co,
```

```
        tableau :
            descr δconstante
              co tableau co
            δconstante ← fairetableau (nbéléments de (collatéral) , modedescr (δ)
              co initialise une constante pour la partie statique du tableau et
                 la range en table co
            affectation (δ, δconstante, statique)
              co génération du code recopiant la partie statique du tableau co
            miseàjourtableau (δ, unparmi (Δ))
              co génération de code pour mettre à jour les triplets de la partie
                 statique du tableau (représentée par δ) et correspondant à un
                 élément (représenté par le descripteur unparmi (Δ)),
                 allouer l'espace pour les éléments du tableau, et
                 mettre à jour l'origine virtuelle co
            pour tout descr δk ∈ Δ
            faire
               vérifbornesetranger (δ, δk, k)
                 co génération de code pour :
                    - vérifier l'égalité des bornes de l'élément (δk) avec les
                      valeurs correspondantes rangées dans les triplets de la
                      partie statique du tableau (δ)
                    - recopier en place l'élément co
            finfaire
                 co fin du cas tableau co,
    fincas
finaction

action miseàjourtableau = (descr δstat, δélem) :
      co δstat : descripteur représentant la partie statique du tableau
         δélem : descripteur représentant un élément du tableau co
    copieaccès (δparambibliothèque, δstat)
    copieaccès (δparambisbibliothèque, δélem)
      co les accès à la partie statique du tableau et à l'élément sont transmis
         dans la zone des paramètres de bibliothèque co
    appelcollat
      co génération de code pour appeler la routine de bibliothèque qui complète
         le descripteur représenté par δstat à partir de l'élément représenté
         par δélem et alloue la place pour les éléments du tableau co
finaction
```

```
action vérifbornesetranger = (descr δstat, δélem, ent k)
     co δstat : descripteur représentant la partie statique du tableau
        δélem : descripteur représentant l'élément à ranger
         k    : rang de l'élément à ranger co
   descr δ ;
   copieaccès (δparambibliothèque, δstat)
   copieaccès (δparambisbibliothèque, δélem)
   δ ← constante (k)
        co génération du code rangeant k en table et initialisation de δ co
   chargeparamètre (δparamterbibliothèque, δ)
        co génération de code transmettant les paramètres co
   appelrecopcollat
        co génération de l'appel à la routine de bibliothèque vérifiant
           l'élément et recopiant cet élément co
finaction

modèle appelcollat =
   (appelbiblio, collatéraltableau, ,)
     co génération de l'appel à la routine de bibliothèque représentée par
        collatéraltableau. Cette routine, dans le cas d'une phrase collatérale
        de tableau, complète la partie statique et alloue l'espace pour les
        éléments co
finmodèle

modèle appelrecopcollat =
   (appelbiblio, recopcollat, ,)
     co génération de l'appel à une routine de bibliothèque recopiant un
        élément de la phrase collatérale de tableau après avoir vérifié sa
        compatibilité avec le tableau co
finmodèle
```

fonction sélectionchamp = (*descr* δbase, *ent* k, *type* modestruct) *descr*

co

δbase : descripteur représentant la structure

k : numéro du champ

modestruct : mode de la structure

allouer et initialiser un descripteur pour qu'il représente la partie statique du $k^{ième}$ champ. Ce descripteur, une fois initialisé à partir du mode de la structure et du descripteur δbase, est retourné comme résultat

co

finfonction

fonction fairetableau = (*ent* nbélem, *type* modetab) *descr*

co

construire et mettre en table des constantes un descripteur de tableau construit à partir du mode précisé par modetab, avec 1 et nbélem, comme bornes inférieure et supérieure de la première dimension. Allouer et initialiser un descripteur pour qu'il représente l'objet possédant cette valeur. Ce descripteur est retourné comme résultat.

co

finfonction

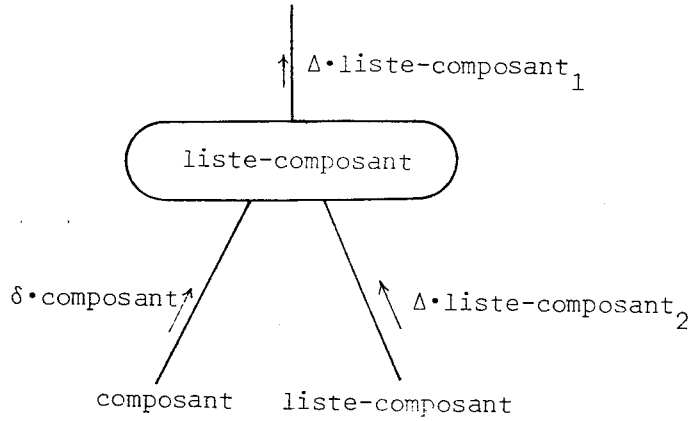
fonction unparmi = (ensemble *descr* Δ) *descr*

co

retourner un descripteur pris au hasard dans l'ensemble Δ

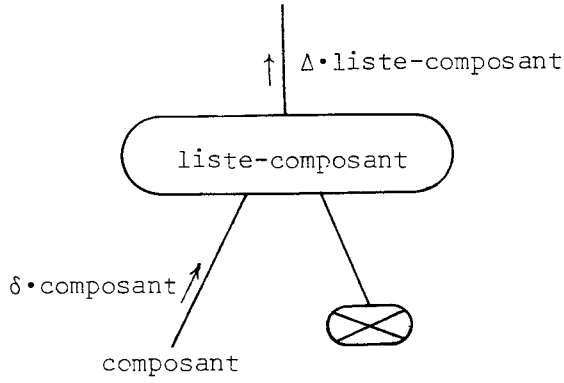
co

finfonction



séquence
  collatéral
    élaborer (composant)
    élaborer (liste-composant)
      co génération de code et synthèse de δ.composant et
        Δ.liste-composant$_2$ *co*
    fincollatéral
    Δ.liste-composant$_1$ ← *δ.composant* + *Δ.liste-composant*$_2$
finséquence

Remarque :
Δ.liste-composant est un ensemble ordonné fini de descripteurs. Le symbole + représente l'opération de concaténation servant à construire cet ensemble.



séquence
  élaborer (composant)
    co génération de code et synthèse de δ.composant co
  Δ.liste-composant ← *δ.composant*
    co création du premier Δ.liste-composant co
finséquence

5.2 - (sélection)

Nous avons déjà signalé deux cas pour l'opération de sélection :

a) la sélection simple qui délivre une valeur dont la représentation est déjà totalement construite et accessible statiquement. La fonction d'accès à la représentation de la valeur sélectionnée peut être obtenue en ajoutant le déplacement (connu statiquement) du champ à la fonction d'accès à la structure.

Pendant la génération de code, disposant du descripteur décrivant la structure, il nous suffit de l'utiliser pour construire celui décrivant la valeur sélectionnée. Ceci ne nécessite aucune production immédiate de code.

Exemple :

début

struct (ent a, réel b) str = ... ;

... a de str ...

...

fin

Soit δbase le descripteur décrivant la base courante. Si d_{str} est le déplacement de la représentation de la structure par rapport à cette base, nous pouvons noter le descripteur décrivant cette valeur structurée par :

$$\delta str = (C_{strer}, (d_{str}, \lfloor \rightarrow \delta base \rfloor))$$

C_{strer} représente les caractéristiques d'une valeur de mode *struct (ent a, réel b)*.

Le traitement de l'opération de sélection *a de str* se résume à construire un descripteur δa représentant la valeur sélectionnée.

$$\delta a = (C_{ent}, (d_{str} + d_a, \lfloor \rightarrow \delta base \rfloor))$$

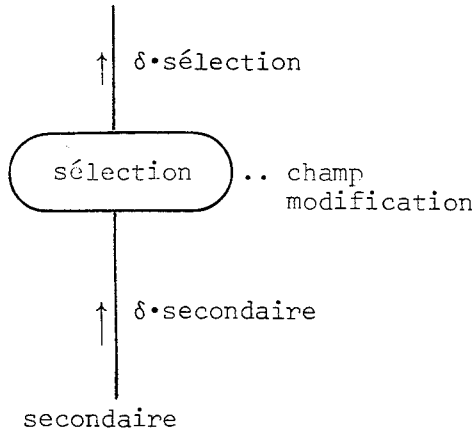
C_{ent} représente les caractéristiques d'une valeur entière

d_a est le déplacement (connu statiquement) de la représentation du champ *a* à l'intérieur de la structure.

En ce point, il n'y a donc pas de production de code.

b) la sélection multiple qui délivre une valeur tableau. Dans ce cas, la valeur sur laquelle s'opère l'opération de sélection étant un tableau, le traitement consiste à produire le code construisant la représentation statique de la valeur sélectionnée, en assurant que la représentation de la partie dynamique correspondante soit accessible. Il n'y a donc pas d'allocation dans la pile dynamique. Remarquons que la représentation de la partie dynamique de la valeur délivrée n'est pas continue.

Remarque :
La théorie voudrait qu'un exemplaire complet (partie statique et partie dynamique) de la valeur soit créé. Les remarques que nous avons faites au sujet de la protection des valeurs et de l'évaluation collatérale (voinch.II.4.1) nous permettent de ne pas le faire et d'éviter ainsi un volume de code non négligeable. Produire du code délivrant les résultats prévus ne pose pas de problèmes sauf dans certains cas très particuliers d'affectation.



décorations

champ : décoration représentant les caractéristiques de la valeur sélectionnée (analogue à une décoration identificateur)

modification : liste des modifications à appliquer à la valeur fournie par l'opération de sélection.

séquence
descr δ
élaborer (secondaire)
co génération de code pour le secondaire co
cas type modedescr (δ.secondaire) dans
tableau :
co sélection multiple co
δ ← allocpse (taille (modech (champ de (sélection))))
δ ← complétermode (δ, modech (champ de (sélection)))
co réservation d'espace pour la partie statique du tableau co
affectation (δ, δ.secondaire, statique)
co génération de code pour copie partie statique du tableau co
majsélectionmultiple (δ, champ de (sélection))
co génération de code pour mise à jour en-tête et origine virtuelle de la partie statique co

```
        sinon
            co sélection simple co
            δ ← modifieraccès (déplacement (champ de (sélection) ), δ.secondai:
            δ ← complétermode (δ, modech (champ de (sélection) ))
    fincas
    gmodif (δ, modification de (sélection) )
    δ.sélection ← δ
finséquence

action majsélectionmultiple = (descr δ, champ champ)
      co génération de code pour mettre à jour :
         a) l'en-tête de la partie statique du tableau à partir du mode de la
         sélection (modech (champ))
         b) l'origine virtuelle du tableau à partir du déplacement de la
         sélection (déplacement (champ)) co
   ranger1 (δ, modech(champ))
      co génération du code modifiant l'en-tête de la partie statique en
         fonction du mode du champ co
   ranger2 (orivirt(δ), constante(déplacement(champ)))
      co génération de code modifiant l'origine virtuelle du tableau d'une
         valeur égale au déplacement du champ dans la structure co
finaction

modèle ranger2 = (descr δdestination, δsource)
      co mise à jour de l'origine virtuelle co
   (+:=, δdestination, δsource)
finmodèle

action ranger1 = (descr δdestination, type mode)
      co
         génération du code modifiant l'en-tête de la partie statique du
         tableau en fonction du mode du champ de la structure
      co
finaction
```

```
fonction modech = (champ ch) type
    co
        délivrer, à partir d'une décoration champ, le mode de l'objet
        correspondant
    co
finfonction

fonction déplacement = (champ ch) ent
    co
        délivrer, à partir d'une décoration champ, le déplacement du
        champ à partir du début de la structure
    co
finfonction
```

5.3 - (tranche)

Remarque :
La description du traitement associé à ce noeud est certainement la plus compliquée parmi les descriptions apparaissant dans cette partie (et donc la plus difficile à suivre) à cause des multiples possibilités offertes à l'utilisateur pour écrire des tranches (bornes et partie *at* optionnelles).

Le but d'une tranche est de délivrer une valeur sous-tableau d'une valeur tableau déjà existante -ce qui n'est guère différent du résultat d'une opération de sélection multiple. Seule la manière de procéder est différente.

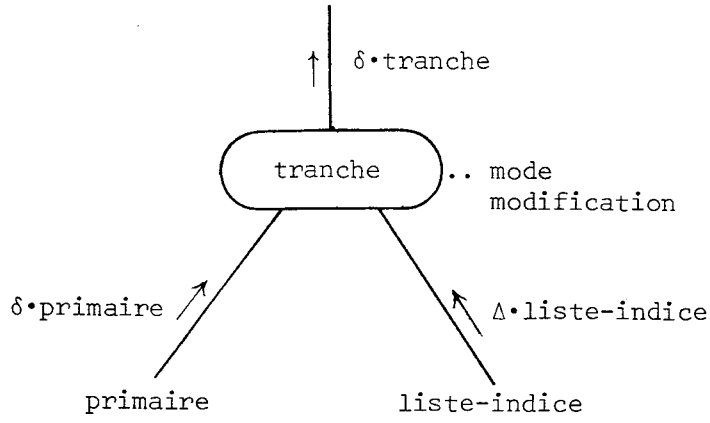
Dans l'arbre abstrait, le noeud (tranche) a deux fils :

- l'un est l'arbre correspondant au primaire (tableau à trancher),

- l'autre est la liste, sous forme explicite, des arbres correspondant aux indices et trancheurs dans l'ordre de leur apparition dans le programme (un trancheur est l'ensemble associé à une dimension de la tranche, et formé de la borne inférieure, de la borne supérieure et de la partie *at*).

Une fois réalisée la génération collatérale du code évaluant le primaire et les éléments de la liste, il faut produire le code construisant la représentation statique. L'initialisation de cette partie statique, qui consiste à calculer les bornes, les enjambées et l'origine virtuelle du tableau résultat, assure l'accessibilité à la représentation de la partie dynamique.

Comme dans le cas de la sélection multiple, il n'y a pas de demande d'allocation dans la zone dynamique car, pour les mêmes raisons, il n'est pas nécessaire de créer un nouvel exemplaire de la partie dynamique de la valeur.



décoration

mode = mode de la valeur tableau délivrée par la tranche

modification = liste des modifications à appliquer à la valeur fournie par l'opération tranche.

séquence

descr δ, δ*constante*

collatéral

élaborer (primaire)

élaborer (liste-indice)

co génération de code pour l'élaboration des bornes et indices de la tranche co

fincollatéral

δ ← *allocpse (taille (mode de* (*tranche*) *))*

δ ← *complétermode (δ, mode de* (*tranche*) *))*

co réservation d'espace pour la partie statique du tableau et construction d'un descripteur co

δ*constante* ← *fairetranche (mode de* (*tranche*) *)*

co initialise une constante pour l'en-tête de la partie statique du tableau et la range en table co

affectation (δ, δconstante, statique)

co génération de code pour initialiser la partie statique du tableau co

miseàjourtranche (δ, δ.primaire, Δ.liste-indices)

co génération de code pour calculer l'origine virtuelle du tableau et initialiser les triplets co

gmodif (δ, modification de (tranche))

δ.tranche ← δ

finséquence

fonction fairetranche = (type mode) descr

co

construit une constante permettant d'initialiser partiellement la partie statique d'un tableau. Les informations nécessaires à cette initialisation sont fournies par le mode de la valeur tableau. La constante est rangée en table et un descripteur est construit pour la décrire

co

finfonction

Les bornes d'une tranche sont représentées pendant la génération de code par des descriptifs (*descrbornes*).

Ces descriptifs sont formés :

- du(des) descripteur(s) habituel(s) décrivant la(les) valeur(s)
- d'une valeur indiquant le type du descriptif

On définit trois types de descriptifs :

descrborne1 = (typeb indice, descr valeurindice)
pour décrire un indice

descrborne2 = (typeb trancheur, descr valeurbi , valeurbs, valeurat)
pour décrire un trancheur. Dans ce cas, les deux premiers descripteurs peuvent être absents

descrborne3 = (typeb bornvide)
pour décrire une absence de bornes (trancheur vide)

Lors du parcours de génération du sous-arbre abstrait représentant les bornes, on construit un ensemble ordonné de descriptifs

(*descrborne* = *union* (*descrborne1* , *descrborne2* , *descrborne3*))

par l'opération + de concaténation habituelle sur les ensembles.

Par suite, on définit sur ces descriptifs :

- les fonctions :

fonction descrbiprésent = (descrbornes δβ) bool
co indique si le descripteur de la borne bi est un descripteur vide ou non co
finfonction

fonction descrbsprésent = (descrbornes δβ) bool
co indique si le descripteur de la borne bs est un descripteur vide ou non co
finfonction

- les fonctions de sélection :

fonction descrindice = (descrbornes δβ) descr
co délivre le descripteur de l'indice contenu dans le descriptif co
finfonction

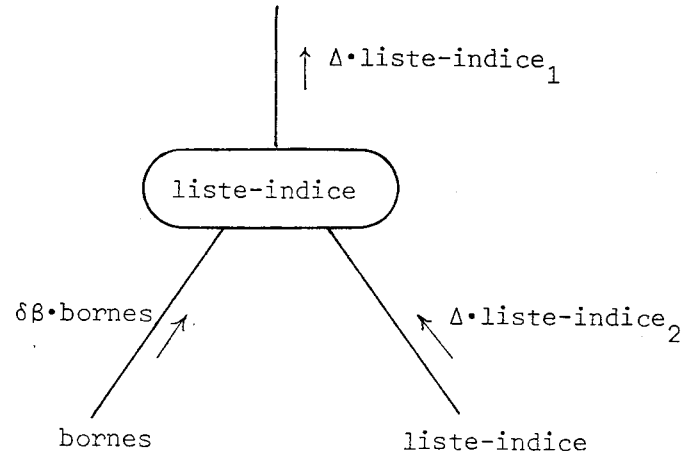
fonction descrbi = (descrbornes δβ) descr
co délivre le descripteur de la borne inférieure contenu dans le descriptif co
finfonction

fonction descrbs = (descrbornes δβ) descr
co délivre le descripteur de la borne supérieure contenu dans le descriptif co
finfonction

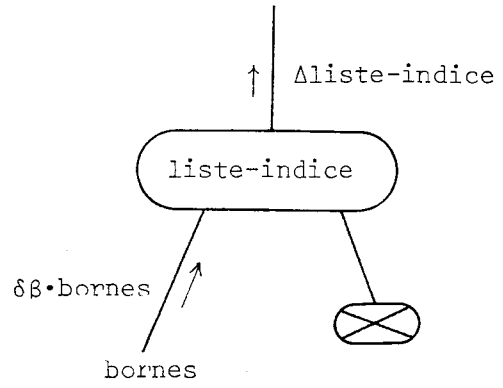
fonction descrat = (descrbornes δβ) descr
co délivre le descripteur de l'origine at contenu dans le descriptif co
finfonction

```
fonction typebornes = (descrbornes δβ) typeb
    co délivre le type du descriptif co
finfonction

action miseàjourtranche = (descrδ, δprim, ensemble descrbornes Δ)
    co   δ      : descripteur de la partie statique du tableau tranche
         Δ      : ensemble des descriptifs de bornes
         δprim  : descripteur du primaire co
    co cette action produit le code qui met en place les triplets et calcule
       la nouvelle origine virtuelle co
  descr δtotalisateur
    allocbase (δtotalisateur, orivirt (δprim))
        co allocation d'une ressource-totalisateur et initialisation de
           cette ressource avec l'origine virtuelle du primaire co
    pour tout descrbornes δβ_k ∈ Δ
    faire
      cas typebornes (δβ_k) dans
        indice : index (descrindice (δβ_k), δtotalisateur, bi (k, δprim),
                 bs (k, δprim), e (k, δprim))
                   co génération de code pour modifier l'origine
                      virtuelle de la tranche co
        trancheur : descr bi, bs
                 bi   si descrbiprésent (δβ_k)
                        alors
                          vérifindice (descrbi (δβ_k),
                                       bi (k, δprim),bs (k, δprim))
                          descrbi (δβ_k)
                        sinon
                          bi (k, δprim)
                        finsi
                   co initialisation du descripteur de la borne inférieure
                      de la dimension courante co
```

```
                bs ← si descrbsprésent (δβ_k)
                        alors
                            vérifindice (descrbs (δβ_k),
                                        bi (k, δprim), bs (k, δprim))
                            descrbs (δβ_k)
                        sinon bs (k, δprim)
                    finsi
                co initialisation du descripteur de la borne supérieure
                    de la dimension courante co
                trancheur (bi, bs, descrat (δβk), e (k, δprim),
                            bi (k,δ), bs (k,δ), e (k,δ), δtotalisateur)
                co génération du code pour initialiser le triplet de
                    la dimension courante et modifier l'origine
                    virtuelle co,
        sinon
            co bornes vides co
            bornes vides (bi (k, δprim), bs (k, δprim), e (k, δprim),
                        bi (k, δ), bs (k, δ), e (k, δ))
            co génération du code recopiant le triplet associé à la dimension
                à partir de celui du primaire co
        fincas
    finfaire
    ranger (orivirt (δ), δtotalisateur)
            co génération de code pour ranger l'origine virtuelle du tableau
                tranche co
finaction
```



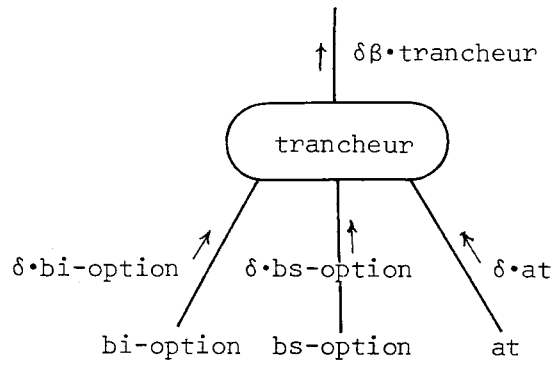
```
collatéral
   élaborer (liste-indice)
   élaborerbornes (bornes, Δ.liste-indice2)
fincollatéral
```




```
élaborerbornes (bornes, )

action élaborerbornes = (arbre bornes, ensemble descrbornes Δ)
     co génération de code pour les bornes associées à une dimension dans
        une tranche et construction des descriptifs de bornes associés co
   si nonnil (bornes)
      alors
         si tranche (bornes)
            alors
                 co la borne est un trancheur co
               séquence
                  élaborer (bornes)
                    co un descriptif de bornes est synthétisé co
                  Δ + δβ.bornes
                    co le descriptif de bornes est concaténé à Δ co
               finséquence
            sinon
                 co la borne est un indice co
               séquence
                  élaborer (bornes)
                    co un descripteur δ.bornes est synthétisé co
                  Δ + consdescrindice (δ.bornes)
                    co le descriptif de bornes est concaténé à Δ co
               finséquence
         finsi
```

```
        sinon
            co borne absente co
            Δ + consdescrbornevide
        finsi
finaction
```

```
fonction tranche = (arbre a) bool
    co
        suivant que la racine de l'arbre est un noeud (trancheur) ou pas
        retourner la valeur "vrai" ou la valeur "faux"
    co
finfonction
```




```
séquence
    descr bi, bs
    collatéral
        bi ← si nonnil (bi-option)
                alors
                    élaborer (bi-option)
                    δ.bi-option
                sinon
                    δvide co descripteur vide co
                finsi
        bs ← si nonnil (bs-option)
                alors
                    élaborer (bs-option)
                    δ.bs-option
                sinon
                    δvide co descripteur vide co
                finsi
```

```
        élaborer (at)
    fincollatéral
    δβ.trancheur ← consdescrtrancheur (bi, bs, δ.at)
finséquence

fonction consdescrindice = (descr δindice) descrborne1
    co
        construction à partir de δindice d'un descriptif d'indice c'est-à-dire
        d'un descriptif de type descrborne1
    co
finfonction

fonction consdescrbornevide = descrborne3
    co
        construction d'un descriptif de trancheur vide c'est-à-dire de type
        descrborne3
    co
finfonction

fonction consdescrtrancheur = (descr δbi, δbs, δat) descrborne2
    co
        construction à partir de δbi, δbs et δat d'un descriptif de trancheur
        de type descrborne2 . δbi, δbs et δat étant les descripteurs représen-
        tant la borne inférieure, la borne supérieure et la partie at du
        trancheur
    co
finfonction

modèle ranger = (descr δdestination, δsource)
    (:=, δdestination, δsource,)
finmodèle

modèle index = (descr δk, δindex, δbi, δbs, δe)
    co cf noeud index co
finmodèle
```

```
modèle trancheur (descr δbi, δbs, δat, δe, δbitranche, δetranche, δ)
     co δbi, δbs, δat : descripteurs des bi, bs, at du trancheur
        δe : descripteur du multiplicateur associé à la dimension
        δbitranche, δbstranche, δetranche : descripteurs des éléments du
        triplet à mettre à jour
        δ' : descripteur du totalisateur pour l'origine virtuelle co
   (:=, δbitranche, δat,)
     co mise en place de la borne inférieure co
   (:=, δtemp, δbi,)
   (-:=, δtemp, δat,)
   (*+:=, δtemp, δe, δ)
     co modification de l'origine virtuelle co
   (:=, δetranche, δe,)
     co mise en place du multiplicateur co
   (:=, δbstranche, δbs,)
   (-:=, δbstranche, δtemp,)
     co mise en place de la borne supérieure co
finmodèle

modèle vérifindice = (descr δindice, δbi, δbs) :
     co cf noeud index co
finmodèle

modèle bornesvide = (descr δbis, δbss, δes, δbid, δbsd, δed) :
     co descripteurs représentant le triplet source :
        δbis, δbss, δes
        descripteurs représentant le triplet destination :
        δbid, δbsd, δed  co
   (:=, δbid, δbis,)
   (:=, δbsd, δbss,)
   (:=, δed, δes,)
finmodèle
```

5.4 - (index)

Le noeud a deux fils :

- l'un est l'arbre correspondant au primaire
- l'autre est la liste, sous forme explicite, des arbres correspondant aux indices, dans l'ordre de leur apparition dans le programme.

Comme pour le noeud (tranche) , le traitement consiste, tout d'abord, à produire collatéralement le code évaluant le primaire et les indices. Le code engendré doit ensuite calculer l'accès à la représentation de la valeur indexée. Pour ce faire, il utilise l'origine virtuelle et les enjambées du tableau ainsi que les valeurs des indices (en vérifiant leur compatibilité avec les bornes).

La représentation de la valeur à laquelle on accède est dans la zone dynamique puisqu'elle appartient à la partie dynamique de la représentation du tableau. L'accès n'est donc pas statique et son calcul a nécessité l'emploi d'une base intermédiaire. Il faut, pendant la génération de code, décrire la valeur fournie par l'indexation. Sachant que son accès sera dans une ressource servant de base, nous construisons deux descripteurs :

δbaseint : décrivant la base intermédiaire nécessaire pour le calcul de l'accès

δindexé : décrivant la valeur fournie.

Dans ces conditions, nous avons :

$\delta_{indexé} = (C_{indexé}, (0, \longrightarrow \delta_{baseint}))$.

$C_{indexé}$ représente les caractéristiques de la valeur retournée par l'indexation.

Nous pourrons ultérieurement modifier statiquement cette fonction d'accès.

Exemple :

début
ent a,b,j,
...
[a:b] struct (ent x, réel y) rstrer = ... ;
...
...x de rstrer [j] ...
...
fin

La valeur retournée par *rstrer* [*j*] est décrite par :

$\delta strer = (C_{strer}, (0, \ \cdot\ ))$ → δbaseint

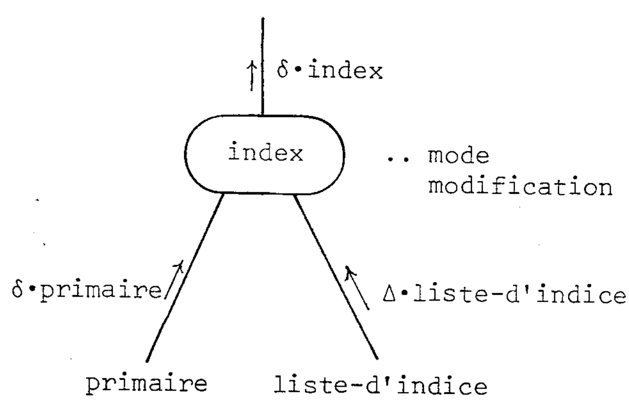
C_{strer} représente les caractéristiques d'une valeur de mode
struct (ent x, réel y)

Le descripteur de la valeur fournie par *x de rstrer* [*j*] est alors :

$\delta x = (C_{ent}, (d_x, \ \cdot\ ))$ → baseint

d_x est le déplacement de la représentation du champ *x* à l'intérieur de la structure.

Remarque :
Dans la pratique, les calculateurs disposent de ressources de type registre pouvant non seulement contenir des adresses mais aussi des valeurs.
Dans ces conditions, une solution alternante consiste à charger dans une de ces ressources la valeur fournie par l'indexation, lorsque cette valeur "tient" dans une telle ressource. Le descripteur décrira alors une valeur en ressource.

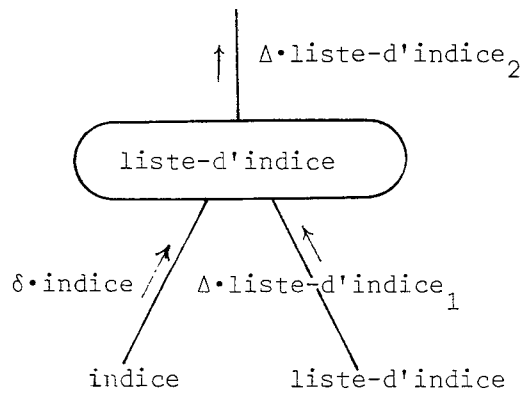


décorations

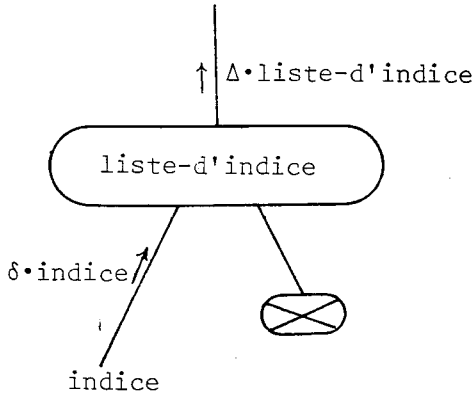
mode = mode de la valeur fournie par l'indexation

modification = liste des modifications à appliquer à la valeur fournie par l'opération d'indexation.

```
séquence
  descr δind, δ
    co allocation d'un descripteur pour l'élément indexé co
    collatéral
      élaborer (primaire)
      élaborer (liste-d'indice)
        co traitement de la liste des indices avec synthèse de
           Δ.liste-d'indice co
    fincollatéral
    calculindex (δ.primaire, Δ.liste-d'indice, δind)
      co génération du code calculant l'adresse de l'élément indexé co
  δind ← complétermode (δind, mode de (index) )
      co compléter le descripteur δind avec la décoration mode pour qu'il
         représente un élement de tableau co
  δ ← allocpse(taille (modedescr (δind))
      co alloue dans la zone d'évaluation un emplacement pour la partie
         statique de l'élément co
  affectation (δ, δind, statique)
      co génération de code pour ranger la partie statique de l'élément
         dans la zone d'évaluation co
  δ ← complétermode (δ, modedescr (δind))
  gmodif (δ, modification de (index) )
finséquence
```




```
séquence
  collatéral
    élaborer (indice)
    élaborer (liste-d'indice)
      co traitement du reste de la
         liste des indices co
  fincollatéral
  Δ.liste-d'indice2 ← δ.élément +
                      Δ.liste-d'indice1
finséquence
```



séquence
élaborer (indice)
Δ.liste-d'indice ← δ.indice
finséquence

action calculindex = (descr δprim, ensemble descr Δliste, descr δ.index)
co δprim = descripteur associé au primaire de l'indexation
Δliste = ensemble des descripteurs associés à la liste des indices
δindex = descripteur résultat co
allocbase (δindex, orivirt (δprim))
co allocation d'une base (de descripteur δindex) et initialisation de cette base avec l'origine virtuelle se trouvant dans la partie statique du tableau représenté par δprim co
pour tout descr δ_k ∈ Δliste
faire
index (δ_k, δindex, bi (k, δprim), bs (k, δprim), e (k, δprim))
finfaire
co δindex représente l'accès à l'élément indexé co
finaction

modèle vérifindice = (descr δk, δbi, δbs)
co δk = descripteur du $k^{ième}$ indice
δbi, δbs = descripteurs des bornes inférieure et supérieure de la $k^{ième}$ dimension co
(option, vérifbornes, ,)
co génération de la vérification de la validité de l'indice co
(comparer, arith, δk, δbi)
(branchement, < , , biblierreurbinf)
(comparer, arith, δk, δbs)
(branchement >, , biblierreurbsup)
(finoption, , ,)
finmodèle

modèle index = (descr δk, δindex, δbi, δbs, δe)
co δk = descripteur de k^ième indice
δindex = descripteur de l'index
δbi, δbs, δe = descripteurs des bornes inférieures, supérieures et de l'enjambée de la k^ième dimension co
vérifindice (δk, δbi, δbs)
co vérification de la validité de l'indice co
(+:=, δk, δe, δindex)*
co génération de la multiplication du k^ième indice par la k^ième enjambée et sommation co
finmodèle

5.5 - Conclusion -

Dans ce chapitre, nous avons traité les constructions du langage créant ou utilisant des valeurs arborescentes.

Certaines de ces constructions sont classiques (indexation, sélection simple) ; d'autres sont des généralisations des constructions habituelles (tranche, collatéral) ; seule l'opération de sélection multiple présente une originalité profonde.

L'utilisateur n'a pas toujours conscience que ce sont ces constructions (excepté la sélection simple et le collatéral neutre), ainsi que l'affectation des valeurs arborescentes qui sont les plus coûteuses en temps d'exécution et en volume de code produit.

6 - NOTATIONS ET VALEURS PAR DEFAUT -

Il existe un certain nombre de constructions du langage qui représentent des constantes ou des valeurs par défaut.

Ces constructions correspondent aux noeuds terminaux de l'arbre abstrait.

(notation) , (fant) , (nil) , (vacuum)

Leurs traitements, bien que particuliers, ne présentent pas de difficultés et font appel à une table des constantes.

6.1 - (notation)

Il s'agit d'une constante apparaissant dans le programme.
Son mode est un mode de base.

Exemples : *1*
3.14

La représentation de la valeur étant dans une table des constantes, il suffit de construire un descripteur représentant cette valeur et l'accès à sa représentation.

Notations de format

En Algol 68, on peut écrire des notations de format. Ces notations sont utilisées dans les opérations d'entrée-sortie avec format et sont associées à l'exécution, à un fichier Algol 68.

Elles peuvent nécessiter dans certains cas une élaboration dynamique, ce qui les distingue des notations habituelles.

Exemple : Considérons la notation de format suivante :

```
$ co exemple de format dynamique co
  co modèle statique pour une valeur entière co
     3d,
  co modèle dynamique pour une valeur entière co
  n (si x co x est une variable co > 0
          alors 4 sinon 6) d
$
```

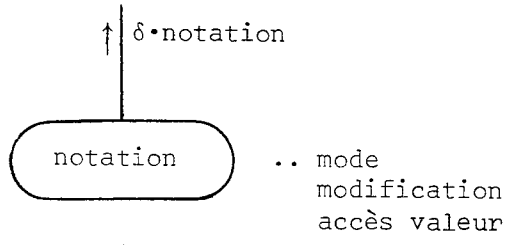
Pour leur traitement à la compilation, on distingue une partie statique, rangée en table comme pour les autres notations, et une partie dynamique représentée sous forme d'un arbre abstrait décoré à partir duquel est produit le code nécessaire à l'élaboration. Ces parties dynamiques sont considérées comme des séquences de code étiquetées par une "étiquette de format", indépendantes et s'élaborant dans l'environnement courant.

Les parties statiques sont formées d'une chaîne codée interprétée lors de l'utilisation du format, à l'exécution, dans les instructions d'entrée-sortie. Il existe des codes particuliers, associés à une étiquette de format, dont l'interprétation provoque l'activation de la séquence de code repérée par cette étiquette. On obtient ainsi, l'élaboration des formats dynamiques. Notons que cette élaboration est définie beaucoup plus simplement, dans la nouvelle version du Rapport, que dans la version précédente [VANWIJ-1], où il était nécessaire de passer par l'étape intermédiaire d'un "transformat".

La compilation de ces formats ne pose pas de problèmes particuliers. (Il est à noter cependant, qu'une notation de format peut contenir une autre notation de format par l'intermédiaire d'une partie dynamique).

Nous donnons dans les notices techniques [68TECH] un exemple de chaîne codée à interpréter pour notre compilateur.

L'utilisateur trouvera cette notion de format inhabituelle et quelque peu complexe à manipuler. Les possibilités offertes sont toutefois grandes : le chapitre 11 du Rapport et le manuel Algol 68 [AFCET-2] en donnent des exemples.



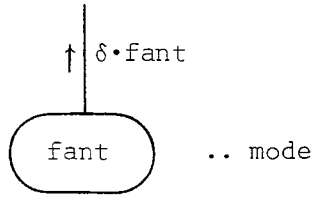
décorations mode : mode de la notation
modificateur : liste des modifications appliquées à la valeur
accès valeur : accès à la valeur de la notation dans la table des constantes.

```
séquence
   descr δ
   δ ← construirenotation
   gmodif (δ, modification de (notation) )
   δ.notation ← δ
finséquence

fonction construirenotation = descr
   descr δ
     co initialiser δ pour qu'il représente la notation co
   δ ← complétermode (δ, mode de (notation) )
   δ ← compléteraccès (δ, accèsvaleur de (notation) )
     co ceci réalise un traitement équivalent à celui réalisé par la
        fonction descridf co
   δ
finfonction  co fin de la fonction construirenotation co

fonction compléteraccès = (descr δ, accès accèsobjet) descr
     co
        compléter le descripteur δ pour qu'il permette de produire le code
        accédant à la valeur qu'il représente. Le descripteur, ainsi
        complété, est retourné comme résultat
     co
finfonction
```

6.2 - (fant)

Ce noeud est associé à la construction Algol 68 *fant* (*skip*).
Il s'agit d'une valeur "fantôme" c'est-à-dire d'une valeur par défaut (non initialisée).
Le traitement consiste à construire dans une table des constantes la représentation d'une valeur fantôme du mode demandé.
Ceci fait, le descripteur représentant cette valeur est retourné comme résultat.

Remarque : On peut utiliser pour cette valeur, la valeur par défaut fournie par certains calculateurs.

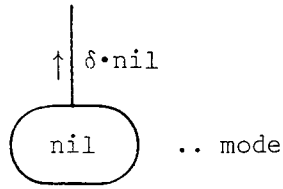


décoration mode : mode de la valeur par défaut.

δ.fant ← fairefantôme (mode de (fant))

6.3 - (nil)

Une valeur pointeur est construite dans une table des constantes et le descripteur représentant cette valeur est retourné comme résultat.



décoration mode : mode de la valeur repère.

δ.nil ← fairenil (mode de (nil))

```
fonction fairenil = (type mode) descr
    co
        construire et mettre en table des constantes une valeur nil du mode
        demandé. Allouer et initialiser un descripteur pour qu'il représente
        cette valeur. Ce descripteur est retourné comme résultat
    co
finfonction
```

6.4 - (vacuum)

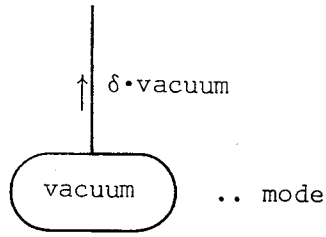
Il s'agit d'une valeur tableau par défaut, la valeur tableau à zéro élément. Une représentation d'une valeur du mode demandé est construite dans une table des constantes et le descripteur la représentant est retourné comme résultat.

Exemple :

```
début
   flex [1:0] ent t ;
   ...
   t := () ;
   ...
fin ;
```

Le tableau t, après l'élaboration de l'affectation t := (), ne comporte plus qu'une partie statique, l'espace éventuellement alloué précédemment pour ses éléments est libéré.

Les principales utilisations de vacuum sont données par les tableaux flexibles.



décoration mode : mode du vacuum. La partie statique du tableau est initialisée à partir de cette décoration.

δ.*vacuum* ← *fairevacuum (mode de* (*vacuum*))

```
fonction fairevacuum : (type mode) descr
    co
        construire et mettre en table des constantes une valeur vacuum du
        mode demandé. Allouer et initialiser un descripteur pour qu'il
        représente cette valeur. Ce descripteur est retourné comme résultat
    co
finfonction
```

6.5 - Conclusion -

Le langage Algol 68 permet d'introduire des valeurs par défaut. La représentation de ces valeurs peut poser quelques problèmes,si le calculateur ne dispose pas de cette possibilité.

L'introduction de ces valeurs peut provoquer quelques aberrations sémantiques telles que :

- effectuer une modification dérepérer sur une valeur *nil*

```
début
   ent i ;
   proc p = rep ent :
      début
      ...
         nil
      fin ;
    i := p
fin
```

- ou effectuer une opération d'indexation ou de tranche par exemple sur une valeur fantôme.
- etc.

Il est par suite nécessaire de vérifier dynamiquement (en engendrant le code adéquat) si ces aberrations ne se produisent pas pour un certain nombre d'opérations, afin d'éviter de poursuivre une élaboration qui n'a plus de sens.

Cette notion de valeur par défaut, introduite en Algol 68, bien qu'insuffisante (le langage n'interdit *nil* et *fant* que s'ils n'apparaissent pas en contexte fort, ce qui évite des écritures comme : *fant* [*i*] ou *nil* := *1*), n'en demeure pas moins intéressante.

La grande richesse du langage nécessite enfin l'introduction de notations inhabituelles comme vacuum ou les notations de format.

7 - AUTRES CONSTRUCTIONS -

7.1 - Affectation -

affectation

Cette construction permet le transfert de valeurs. Le noeud est formé de deux opérandes :

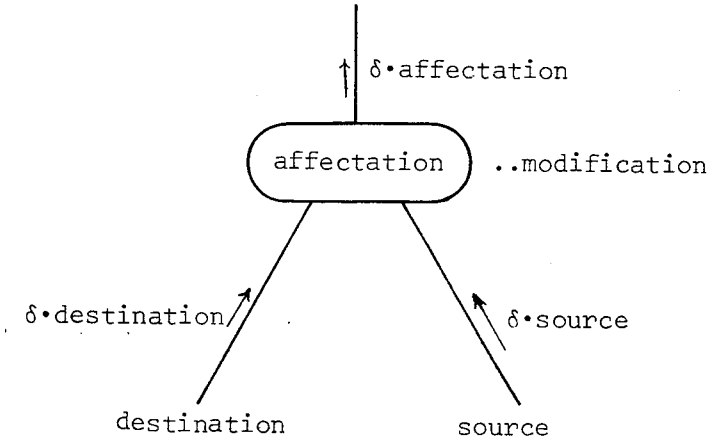
- l'arbre correspondant à la destination (partie gauche de l'affectation)
- l'arbre correspondant à la source (partie droite de l'affectation).

Le traitement se réduit à la génération du code évaluant la source et la destination, suivi de celui réalisant le transfert de la représentation de la valeur.

Cette action peut nécessiter un volume important de code, suivant le mode de la valeur à affecter.

Si la valeur n'a pas de partie dynamique, l'affectation est simple à réaliser car la représentation de la valeur est toujours continue.

Si la valeur possède des parties dynamiques, le code produit doit parcourir les représentations arborescentes de la source et de la destination. Durant ces parcours, pour chaque tableau à affecter, le code produit doit aussi vérifier la compatibilité des tableaux source et destination (notamment l'égalité des bornes).



décoration

modification = liste des modifications à appliquer à la valeur fournie par l'affectation.

séquence
collatéral
élaborer (source)
co génération de code évaluant la source co
élaborer (destination)
co génération de code évaluant la destination co
fincollatéral
affectation (δ.destination, δ.source, statdyn)
gmodif (δ.destination, modification de (affectation))
δ.affectation ← δ.destination
finséquence

Remarque :

D'après les considérations faites sur l'élaboration collatérale dans les chapitres précédents, l'affectation d'une tranche de tableau peut poser des problèmes suivant la stratégie adoptée pour la protection des valeurs et la création des exemplaires de valeurs.

Rappelons l'exemple déjà cité :

```
début
   [1:4] ent t := ... ;
   t[2:4] := t[1:3] ;
   ...
fin
```

Si nous ne prenons aucune précaution, les éléments du tableau risquent d'être tous égaux (à *t[1]*) après l'affectation.

C'est pourquoi il est nécessaire, si l'on désire éviter cet "effet de bord", de produire du code supplémentaire avant celui de chaque affectation d'une valeur tableau. Ce code est chargé de vérifier que les représentations des tableaux source et destination ne se recouvrent pas. En cas de recouvrement il faut alors protéger l'une des deux valeurs par création d'un nouvel exemplaire.

7.2 - Forceur -

forceur

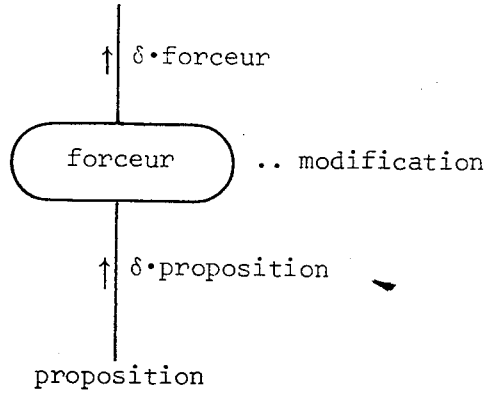
Cette construction permet l'utilisation de toutes les modifications sur la valeur forcée.

Le traitement consiste à engendrer le code évaluant la valeur à soumettre aux modifications.

Exemple :

```
début
   rep rep ent rre = loc rep ent := loc ent ;
   rep ent (rre) := 1 ;
   ...
fin
```

Le forceur *rep ent (rre)* permet de dérepérer *rre*. On obtient ainsi une valeur de mode *rep ent* à laquelle on peut affecter la valeur *1*.



décoration

modification = liste des modifications à appliquer à la valeur fournie par le forceur.

séquence

élaborer (proposition)

gmodif (δ.proposition, modification de (*forceur*))

δ.forceur ← δ.proposition

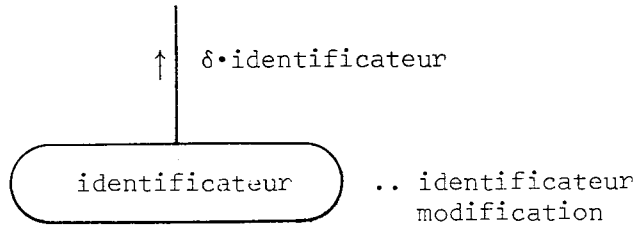
finséquence

7.3 - Utilisation d'identificateur -

(identificateur)

Nous construisons un descripteur de compilation représentant la valeur possédée par l'identificateur.

Comme pour les autres constructions délivrant des valeurs, il n'est pas nécessaire de créer un nouvel exemplaire de la valeur.



décorations

identificateur = décoration décrivant les caractéristiques de la valeur possédée par l'identificateur

modification = liste des modifications à appliquer à la valeur fournie.

séquence

descr δ

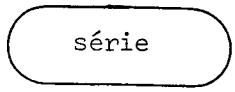
δ ← descridf (identificateur de (*identificateur*))

gmodif (δ, modification de (*identificateur*))

δ.identificateur ← δ

finséquence

7.4 - Phrase sérielle -

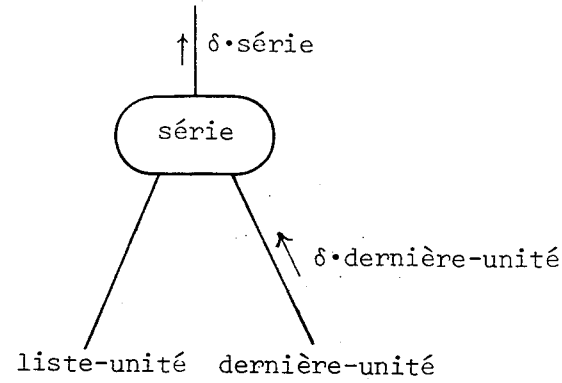


Ce noeud à deux fils :

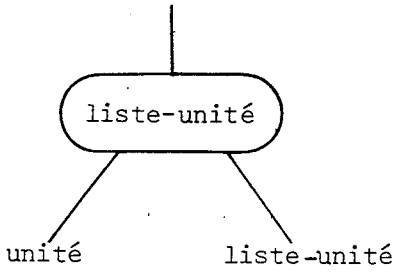
- l'un sous forme explicite de liste. Cette liste est constituée de l'ensemble des arbres correspondant aux unités constituantes de la phrase sérielle, excepté celui de la dernière unité. Ces unités sont à évaluer séquentiellement.

- l'autre est l'arbre de la dernière unité qui est celle délivrant la valeur de la phrase sérielle.

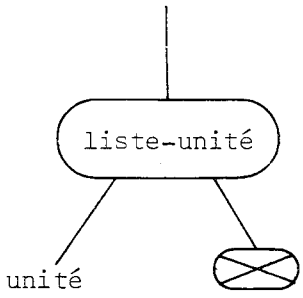
Le traitement consiste à produire séquentiellement le code évaluant les unités de la liste puis celui évaluant la dernière unité.



séquence
élaborer (liste-unité)
élaborer (dernière-unité)
δ.*série* ← δ.*dernière-unité*
finséquence



séquence
élaborer (unité)
élaborer (liste-unité)
finséquence



élaborer (unité)

7.5 - Conclusion -

Les noeuds de ce chapitre n'ont pas de points commun entre eux. Ils ont simplement le mérite de terminer la description noeud par noeud de la phase de génération de code ! Signalons toutefois, l'intérêt du forceur qui permet d'appliquer l'ensemble des modifications existantes à une valeur dont la position syntaxique restreint les possibilités d'application automatique des modifications.

La description de la génération de code nous a permis de mettre en relief certains aspects du langage Algol 68.
S'il présente, pour l'utilisateur, des possibilités nouvelles par rapport aux autres langages de programmation, elles nous ont posé parfois, des problèmes de génération de code. Ce fut le cas par exemple des choix union ,etc ... Malgré cela, le code produit reste efficace, ce qui justifie l'intérêt que l'on peut porter à ce langage. Un implanteur a, bien sûr, des critiques à formuler sur sa définition et notamment sur l'aberration sémantique introduite dans certaines constructions par le souci constant d'orthogonalité de la définition. Mais, ce ne sont là que des griefs mineurs.

Ce n'est d'ailleurs pas tellement sur la généralité de ses constructions mais plutôt sur les possibilités d'utilisation de structure de données très variées, que le langage Algol 68 se différentie des autres langages de programmation. En effet, la notion de mode et les règles de construction des déclareurs de mode permettent à l'utilisateur de représenter et de manipuler ses données de façon plus naturelle et plus concise.

Exemple :

```
début co produit de deux nombres complexes co
   mode polaire = struct (réel ρ, θ) ;
   polaire z1 := ... ,  co z1 = (ρ1, θ1) co
         z2 := ... ,  co z2 = (ρ2, θ2) co
          z ;
   op * = (polaire t1, t2) polaire :
      (ρ de t1 * ρ de t2, θ1 de t1 + θ2 de t2) ;
   ...
      z := z1 * z2
fin
```

Dans l'implantation d'un langage quelconque, la phase de génération, pour remplir parfaitement son rôle, doit voir sa réalisation précédée d'une étude approfondie. A cause de la puissance du langage, cette étude devient fondamentale dans le cas d'Algol 68.

Indépendamment de son intérêt purement descriptif, la description sous forme algorithmique s'est révélée particulièrement bien adaptée à une telle étude.

II.5 - CONCLUSION -

ASPECTS DECLARATIFS ET ALGORITHMIQUES ASSOCIES A LA DESCRIPTION

Le principal outil utilisé dans cette description, est constitué par les attributs sémantiques qui permettent, entre autres, de mettre en évidence le principe de localité de la traduction. Les attributs utilisés, définis sur une arborescence, sont une adaptation des attributs habituellement définis sur une grammaire hors-contexte. Cette adaptation et les notations qui lui sont associées ont été présentées précédemment (voir ch. II.1, § 1).

Nous avons été amenés, au niveau de la description, à distinguer deux parties dans la traduction pour qu'elle puisse remplir les objectifs que nous lui avons assignés. La première partie comprend la construction de l'arbre abstrait primitif et de ses décorations, la seconde la production de code.

1 - L'aspect déclaratif introduit par les attributs convient parfaitement à la description de la première partie de la traduction. On peut remarquer, que cette description

- reste assez proche de la description formelle du langage
- peut servir de spécifications pour une implantation raisonnable.

Si l'on dispose d'un système d'attributs, elle peut servir également de base à une définition complète du compilateur à défaut d'une réalisation. On peut alors vérifier la cohérence de cette définition, ce qui constitue un pas important vers la validation d'un compilateur [LORHO].

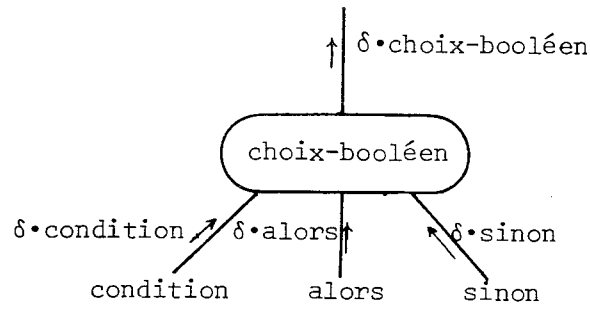
2 - Dans la deuxième partie nous avons défini et utilisé de nouveaux outils adaptés à la description de la génération de code. Malgré certaines ressemblances, ces outils diffèrent profondément des attributs sémantiques. Une comparaison permet de mieux comprendre ces différences.

Commençons par examiner ce qu'il en est pour la description de la génération de code (à partir de l'arbre abstrait décoré) si nous n'utilisons que des attributs sémantiques. Nous considérons, à titre d'exemple, le nœud (choix-booléen) .

Les notations utilisées ici sont les mêmes que dans la description (voir ch. II.4.6 § 1).

Dans une première approche, nous ne tenons pas compte des contraintes de gestion statique à la compilation des ressources machine (registres ou accumulateurs, mémoire, etc...) nécessaires à l'exécution du programme. Cette abstraction nous permet de nous rapprocher de la définition du langage où le calculateur n'est qu'"hypothétique".

On peut alors schématiser le noeud (choix-booléen) et les descripteurs de valeur associés à chaque branche comme suit :



Dans ces conditions, le code engendré pour les trois branches du choix peut être le suivant :

a) code pour la branche condition :

génération du code pour élaborer la partie condition,
puis génération du code décrit par les quadruplets :
(comparer, bool, δvrai, δcondition)
(branchement, faux, étiquette1,).

Cet ensemble de quadruplet est noté, dans la suite, codechoix1.

b) code pour la branche alors :

génération du code pour élaborer la partie alors,
puis génération du code décrit par les quadruplets :
(branchement, inconditionnel, étiquette2,)
(defétiq, étiquette 1, ,)

Cet ensemble de quadruplets est noté, dans la suite, codechoix2.

c) code pour la branche sinon :

génération du code pour élaborer la partie sinon,
puis génération du code décrit par le quadruplet
(defétiq, étiquette2, ,) noté dans la suite codechoix3.

Le descripteur de valeur δ•choix-booléen représentant la valeur-résultat est construit à partir du descripteur δ•sinon (ou δ•alors). Pour décrire la génération de code, l'utilisation d'attributs est bien adaptée, car il n'apparait aucune contrainte dans l'arbre.

On obtient ainsi, si l'on introduit un attribut synthétisé, *code.sélecteur*, sur chaque branche une description dont nous ne précisons ici que les règles de calcul des attributs. (Ce nouvel attribut *code* représente le code engendré).

Les règles de calculs associées au noeud sont :

$$\delta\text{•}choix\text{-}booléen \leftarrow construirerésultat\ (\delta\text{•}sinon)$$
$$code\text{•}choix\text{-}booléen \leftarrow code\text{•}condition + codechoix1\ (\delta\text{•}condition) + code\text{•}alors + codechoix2\ (\delta\text{•}alors) + code\text{•}sinon + codechoix3\ (\delta\text{•}sinon)$$

où : - *construirerésultat* est une fonction construisant l'attribut synthétisé (descripteur de valeur).

- l'opération + permet de concaténer deux séquences de code (cette concaténation garde l'aspect séquentiel de l'exécution du code)

- *codechoix1*, *codechoix2*, *codechoix3* sont des fonctions permettant de produire le code décrit par les ensembles de quadruplets définis ci-dessus aux paragraphes a, b, c.

La description ainsi obtenue, quoique incomplète, reste simple, et proche de la définition formelle du langage. Si un tel niveau de description était suffisant pour la description de la construction de l'arbre décoré, il n'en est plus de même pour la génération de code.

Les notions de calculateur (ou de classe de calculateurs) et de ressources, signalées précédemment, bien qu'absentes de toute définition formelle, sont essentielles pour préciser les spécifications complètes d'une implantation. La gestion statique des ressources de calcul (registres par exemple) et mémoire est une des principales difficultés de la génération de code.

Les ressources de calcul (ou de mémoire) sont intimement liées au code engendré puisque la plupart des instructions-machine ont des opérandes utilisant ces ressources. Il existe donc une dépendance entre l'environnement d'élaboration d'un programme et le code engendré pour cette élaboration (ce code est amené à modifier cet environnement). Nous avons été ainsi conduits à préciser, avant la description de la génération de code, la structure de cet environnement (voir ch. II.4.1). Dans notre cas cet environnement est constitué notamment par une pile classique, représentée de manière continue (pour des raisons d'efficacité sur la classe de machines à laquelle nous nous intéressons).

Examinons de nouveau le cas de l'instruction choix-booléen. Il nous faut tout d'abord préciser certains aspects sémantiques du langage concernant la gestion des environnements.

Soit l'instruction Algol 68 :

si (*ent* a, b ; ... a > b)
alors (*ent* c ;...a ; c + := a ; ...)
sinon (*ent* d ;...a ; d + := a ; ...)
fsi

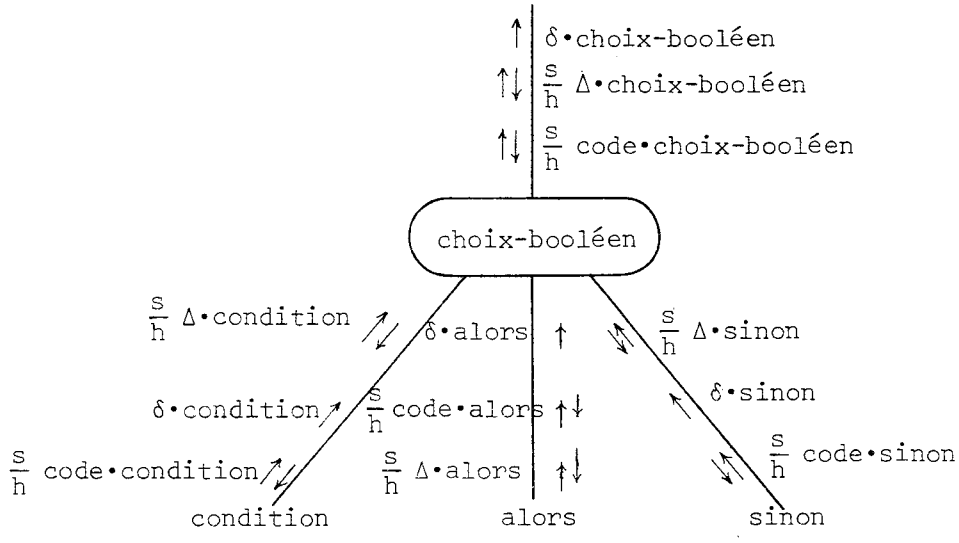
l'objet dont le nom externe est *a*, est accessible depuis l'alternance *alors* ou *sinon*, alors que l'objet identifié par c n'est pas accessible dans la partie condition ni dans l'alternance *sinon*. On peut remarquer qu'il faut produire le code pour les deux alternances du choix (contrairement à un processus d'interprétation), bien qu'une seule des deux soit nécessaire à l'exécution. La gestion de l'environnement pour l'instruction choix booléen, conduit à engendrer le code de la condition préalablement à celui des deux alternances.

En effet, l'exécution du code associé à la condition mettra en place l'environnement dans lequel pourra s'élaborer la partie *alors* ou *sinon*. En conséquence, il n'est pas possible de produire concurremment le code pour les trois branches du noeud (choix-booléen) en leur faisant partager les mêmes ressources (car cela conduirait à utiliser de manière incohérente ces ressources pour constituer cet environnement). Dans ces conditions, il n'est plus suffisant de considérer le code comme un attribut uniquement synthétisé dont la construction pour le noeud (choix-booléen) se ferait par concaténation des séquences de code synthétisées par ses fils. Nous avons ainsi

introduit une contrainte externe à l'arbre abstrait, qui peut se traduire, dans la description par attributs, par l'utilisation d'un mécanisme de synchronisation permettant d'engendrer complètement le code sur la branche condition avant celui des autres. Cette synchronisation peut être mise en place en transformant l'attribut code synthétisé en attribut mixte (synthétisé et hérité). Nous introduisons également un autre attribut mixte décrivant les ressources statiques, utilisées à l'exécution, et à partir desquelles est constitué l'environnement d'exécution.

On obtient la description suivante pour le noeud (choix-booléen) :

code est l'attribut mixte décrivant le code engendré. Dans la pratique, il peut ne représenter qu'un déplacement courant dans une zone tampon (de code objet) pour éviter de manipuler des séquences de code.
Δ est l'attribut mixte "environnement" qui permet de décrire les ressources à l'exécution gérées à la compilation. Dans notre cas, par exemple, les registres sont utilisés comme ressources de calcul ou comme base pour décrire l'accès à un objet et sont nécessaires au codage des instructions. Au moment de la génération de code, la gestion des ressources-mémoire se ramène à celle de la zone d'évaluation courante, du vecteur d'accès aux objets non locaux et de la base de l'environnement courant. L'attribut environnement Δ est donc essentiellement constitué d'un ensemble de descripteurs décrivant les ressources registres et les éléments du vecteur d'accès aux objets non locaux.



Les règles de calcul des attributs *code* et environnement Δ sont de la forme :

hcode•condition ← hcode•choix-booléen
hΔ•condition ← hΔ•choix-booléen
(Δ1, code1) ← bchoix1 (δ•condition, sΔ•condition)
co bchoix1 est une fonction a deux paramètres délivrant deux valeurs : l'attribut environnement : Δ1 et le code : code1 permettant de produire le code nécessaire à la comparaison et au branchement co
hcode•alors ← scode•condition + code1
hΔ•alors ← Δ1
(Δ2, code2) ← bchoix2 (δ•alors, sΔ•alors)
co bchoix2 est une fonction de génération délivrant deux résultats comme bchoix1 co
hcode•sinon ← scode•alors + code2
hΔ•sinon ← Δ1
co hΔ•sinon est identique à hΔ•alors co
(Δ2, code3) ← bchoix3 (δ•sinon, sΔ•sinon)
co bchoix3 est une fonction de génération analogue à bchoix1 ou bchoix2 co
co l'environnement décrit par Δ2 est le même après la partie alors ou la partie sinon co
scode•choix-booléen ← scode•sinon + code3
sΔ•choix-booléen ← Δ2

Le descripteur de valeur synthétisé est calculé par :

δ•choix-booléen ← construirerésultat(δ•sinon)

Remarque : on ne tient pas compte dans cette description de la gestion proprement dite des ressources, ni de la production du code, qui s'effectuent par l'intermédiaire des fonctions bchoix1, bchoix2, bchoix3.

La description ainsi obtenue permet, comme celle de la construction de l'arbre abstrait décoré, de réaliser les objectifs fixés, en autres : servir de spécification à une implantation "raisonnable". Nous pouvons, toutefois remarquer que nous avons fixé arbitrairement un ordre d'évaluation des attributs des branches *alors* et *sinon*. Dans le cas présent nous avons choisi la solution la plus naturelle.

De tels choix peuvent dans d'autres cas, noeud (index) par exemple, être décidés en fonction de critères d'optimisation qui ne doivent pas intervenir dans cette description,mais plutôt au niveau de sa mise en oeuvre pour réaliser un compilateur. Notre but n'étant pas d'écrire ce compilateur à l'aide d'un système d'attributs il nous a semblé dommage de supprimer cette étape importante dans le "raffinement" qui conduit à l'implantation effective. C'est pourquoi nous avons préféré abandonner la forme déclarative de la description pour retourner à une forme impérative. Nous avons cependant essayé de garder certains aspects intéressants de la description par attributs. La localité de la traduction intervient encore au niveau des descripteurs de valeur ; les attributs mixtes, *code* et Δ apparaissant sur tous les noeuds ont été transformés en variables globales,non seulement pour simplifier, mais pour être plus conformes à la réalité des implantations. En contrepartie les structures de contrôle algorithmique introduites, font disparaître partiellement la localité et les possibilités de vérification de cohérence.

La description algorithmique proposée est faite en termes d'actions et d'objets Certains objets, correspondant aux (attributs synthétisés) descripteurs de valeurs, ont une forme inhabituelle. Cette description nous apparaît plus simple (la synchronisation n'apparaît plus artificiellement) et plus lisible ; nous montrons dans la troisième partie qu'elle permet d'obtenir assez facilement un compilateur raisonnable en poursuivant le raffinement. D'autre part, elle permet d'exploiter toutes les solutions possibles pour effectuer des optimisations. On peut toujours la transcrire, en une description par attributs en mettant en oeuvre systématiquement les attributs mixtes *code* et Δ (environnement) utilisés dans l'exemple précédent et en fixant un ordre chaque fois que l'on a un choix dans le parcours des fils (action de contrôle *collateral* ... *fincollatéral)*.

III - REPRESENTATION DE LA FONCTION DE TRADUCTION

III.1 - INTRODUCTION -

Le rôle de cette partie est de montrer comment on peut effectivement produire un compilateur à partir de la description qui vient d'être faite de la fonction de traduction. Nous présentons ici des choix qui n'interviennent pas dans la description, comme une méthode particulière d'analyse syntaxique ou la production effective de code à partir de quadruplets. Cette partie constitue un niveau intermédiaire entre la description et les notices techniques. Elle constitue également une introduction aux notices techniques du compilateur, qui n'apparaissent pas dans cette thèse.

Nous présentons d'abord l'architecture de notre compilateur, puis nous abordons les points suivants :

- la représentation effective de l'arbre abstrait, ainsi qu'une méthode de segmentation, qui rend raisonnable son utilisation dans un compilateur,
- la représentation des décorations, le processus d'équivalence des modes et la fonction "modifier",
- le processus d'analyse syntaxique, sous-jacent à la construction de l'arbre abstrait,
- la production de code machine à partir des quadruplets utilisés dans la description. On précise également la représentation des objets à l'exécution, et la gestion des ressources du calculateur, qui est implicite dans la description. On montre comment certains choix pratiques qui conduisent à une plus grande efficacité, sont rendus possibles par l'emploi dans la description, du schéma de contrôle *collatéral* ... *fincollatéral* .

De nombreux aspects ne sont pas traités ici. Citons l'implantation du parallélisme, qui fait l'objet d'une étude particulière [68GRE-8]. On ne trouve pas non plus la description détaillée des structures de données utilisées dans le compilateur, la grammaire d'analyse, le traitement des entrées-sorties, ou le manuel d'utilisation du compilateur. Ces points font l'objet de notices techniques ou de rapports. Nous avons voulu éviter de donner cette foule de détails ou de recettes particulières qui fleurissent souvent dans un compilateur, en nous limitant aux aspects que nous jugeons les plus importants.

III.2 - ARCHITECTURE DU COMPILATEUR -

La première caractéristique que l'on donne habituellement d'un compilateur est le nombre de passages qu'il réalise. Cette information n'est pas toujours significative. Dans notre cas, nous avons un compilateur en trois passages. En effet, nous avons deux textes intermédiaires, images du texte source, qui existent chacun, en entier, à un moment donné du processus de compilation, délimitant ainsi trois passages. Bien que cette notion ne soit pas sans intérêt pour la connaissance de notre implantation, nous pensons qu'elle masque l'essentiel, qui est sa conception autour de l'arbre abstrait. Nous verrons que cet arbre abstrait décoré n'existe jamais dans sa totalité à un instant donné. Il ne peut donc être qualifié de texte intermédiaire délimitant deux passages.

Pour comprendre un découpage du traitement en plusieurs phases ou passages, examinons le programme suivant :

```
début
    réel x ; ent n ;
    proc p = ... : (m x ;                ...   x   ...   )
    mode m = [1 : n] ent ;
    ...
fin
```

Dans le corps de la procédure *p*, *m x* est une déclaration de l'identificateur x comme étant un tableau d'entiers. Or, dans une exploration du texte de gauche à droite, *m* n'est pas connu au moment où l'on rencontre *m x*, qui pourrait aussi bien être une formule (comme *abs x*) si *m* était déclaré comme opérateur. Dans l'hypothèse d'un découpage du traitement en passages, avec exploration du texte de gauche à droite sans retour arrière, on est conduit à deux passages pour traiter le problème des indicateurs. Le premier construit les ensembles des déclarations d'indicateurs par région et le second réalise leur identification.

La connaissance du mode des identificateurs nécessite le même découpage, car une utilisation d'identificateur peut apparaître lexicographiquement avant sa déclaration, et le traitement des modifications nécessite la connaissance du mode de tous les identificateurs. La production de code ne peut s'effectuer qu'après le traitement des modifications. On est conduit au schéma suivant :

1) - prise en compte des déclarations d'indicateur

2) - identification des indicateurs

3) - prise en compte des déclarations d'identificateur et d'opérateur

4) - identification des identificateurs

5) - traitement des modifications et identification des opérateurs

6) - production de code.

En regroupant les traitements qui peuvent être faits simultanément, on est conduit au schéma en quatre passages proposé par Mailloux [MAILLOUX] : 1), puis 2), 3), puis 4), 5) et 6).

On retrouve dans notre implantation la même séquence de traitements. Ils sont regroupés en trois passages dont nous donnons maintenant le schéma.

1 - SCHEMA D'ORGANISATION DU COMPILATEUR -

1 - Premier passage :

1.1 - Lecture de gauche à droite du texte source et reconnaissance de la structure par région.

1.2 - Construction des tables d'indicateurs de mode et de priorité pour chaque région.

1.3 - Marquage du type des parenthèses et des déclareurs.

1.4 - Mise en table des modes apparaissant dans les déclarations de mode.

texte intermédiaire : - image codée du texte source
- tables

1.5 - Processus d'équivalence des modes apparaissant dans les déclarations de mode. Construction d'un représentant unique pour les modes équivalents.

2 - Deuxième passage :

2.1 - Analyse syntaxique hors-contexte selon un schéma LL(1).

2.2 - Identification des indicateurs de mode et de priorité.

2.3 - Construction des tables d'identificateurs et d'opérateurs pour chaque région.

2.4 - Construction de l'arbre primitif partiellement décoré et segmentation de cet arbre.

texte intermédiaire : - arbre primitif segmenté, sous forme postfixée, partiellement décoré.
- tables.

3 - Troisième passage :

Pour chaque "constituant" du texte postfixé.

3.1 - Construction d'un arbre avec pointeurs explicites.

3.2 - Identification des identificateurs.

3.3 - Traitement des modifications et identification des opérateurs.

3.4 - Génération de code.

texte intermédiaire : code machine chargeable.

Avant chaque exécution, un chargeur spécialisé transforme le code chargeable en code exécutable, résolvant au passage quelques problèmes non traités pendant la phase de production de code (étiquettes, adressage propre au calculateur IBM 360).

Ce schéma diffère de celui proposé par Mailloux, au niveau du troisième passage. Si nous n'avions pas réalisé une segmentation de l'arbre abstrait, nous aurions eu l'organisation suivante, qui s'en rapproche plus :

- le texte intermédiaire produit par le deuxième passage serait directement l'arbre primitif partiellement décoré, représenté sous sa forme définitive, c'est-à-dire avec pointeurs explicites.
- le troisième passage réaliserait l'identification des identificateurs, le traitement des modifications et l'identification des opérateurs. A ce point, nous aurions donc l'arbre abstrait décoré tel qu'il apparaît dans la description, avant la production de code.
- le quatrième passage réaliserait la production de code.

Pour rattacher le schéma d'organisation du compilateur à la description de la fonction de traduction, nous montrons les traitements qui sont réalisés à chaque étape de la traduction, telle qu'elle est décrite dans la deuxième partie.

2 - RELATION AVEC LA DESCRIPTION -

- Le point de départ de la description (chapitre II.3.1 § 1).
  Nous avons supposé résolus certains problèmes concernant Algol 68, par un traitement préalable du texte source. Ce traitement correspond aux points 1.1, 1.2, 1.5 et 2.2 du schéma précédent.

- La construction de l'arbre primitif (chapitre II.3.2).
  Le processus d'analyse syntaxique sous-jacent à cette construction est réalisé en 2.1. La construction de l'arbre primitif est décomposée en la production d'un texte postfixé (2.4) et la construction d'un arbre avec pointeurs explicites pour chaque "constituant" du texte postfixé (3.1).

- La première phase de décoration : l'identification des identificateurs (chapitre II.3.3).
  Ce processus comprend la construction des tables d'identificateurs par région (2.3) et l'association effective des noms (3.2).

- La deuxième phase de décoration : le traitement des modifications et l'identification des opérateurs (chapitre II.3.4).
  Ce traitement correspond exactement à 3.3.

- La production de code (chapitre II.4)
  Ce traitement correspond à 3.4.

Les différences entre l'organisation du compilateur et la description de la traduction sont dues au fait que dans la description nous avons fait un découpage "logique" des traitements. En pratique, certains de ces traitements ont dû être décomposés et répartis sur deux passages. C'est le cas de l'identification des identificateurs, qui correspond à un seul traitement dans la description. Dans la réalisation, la construction des tables est faite au deuxième passage et l'association effective des noms au troisième passage. D'autres traitements ont été regroupés dans un même passage. C'est le cas du traitement des modifications et de la production de code, qui sont réalisés au troisième passage. Ce regroupement a été rendu possible grâce à l'emploi d'une méthode de découpage de l'arbre abstrait, que nous présentons maintenant.

III.3 - REPRESENTATION DE L'ARBRE - SEGMENTATION -

1 - PRINCIPE D'UNE SEGMENTATION -

Le deuxième passage produit l'arbre abstrait sous forme postfixée. Le traitement des modifications et la génération de code s'effectuent sur l'arbre abstrait représenté avec des pointeurs vers les opérandes. Afin d'éviter la présence de la représentation avec pointeurs explicites de tout l'arbre abstrait, en mémoire, le texte postfixé est découpé d'une manière telle qu'il suffit pour les traitements ultérieurs de n'en disposer que d'une petite partie.

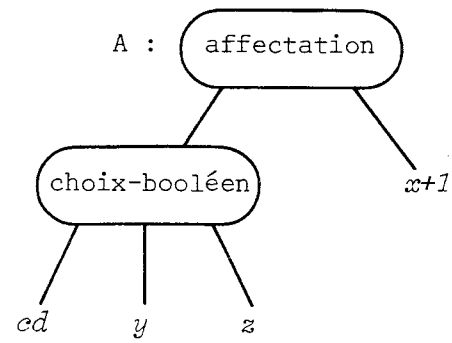
Cette méthode [68GRE-5] est basée sur une propriété des propositions sérielles la valeur d'une proposition sérielle est celle de sa dernière unité constituante

Quelles sont les valeurs à prendre en compte lors du traitement des modifications de la phrase suivante ?

(1) *si cd alors y sinon f(z) ; z fsi := (réel x ; lire(x) ; x+1)*

P.S.$_1$ P.S.$_2$ P.S.$_3$ P.S.$_4$

Dans chaque proposition sérielle - éventuellement réduite à une unité -, seule la valeur de la dernière unité est prise en compte. Les autres unités sont "neutralisées". Ainsi, dans ce cas, on peut considérer l'arbre simplifié A.



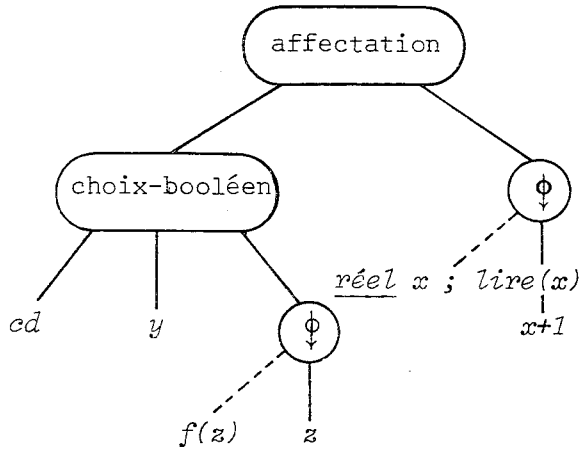
Il suffit au traitement des modifications, qui consiste à :
- s'assurer que *cd* est booléen,
- vérifier que *y* et *z* sont des références à un même mode,
-vérifier que *x+1* délivre une valeur de ce mode, tout en émettant les modifications adéquates si nécessaire. Voilà l'arbre simplifié dont nous aimerions disposer pour le traitement des modifications. Il faut remarquer que cet arbre reste le même quel que soit le nombre d'unités qui composent les 4 propositions sérielles. Par exemple pour :

(2) *si u1 ; u2 ; u3 ; (cd) alors u4 ; u5 ; u6 ; (y) sinon f(2) ; (z) fsi := (...)*

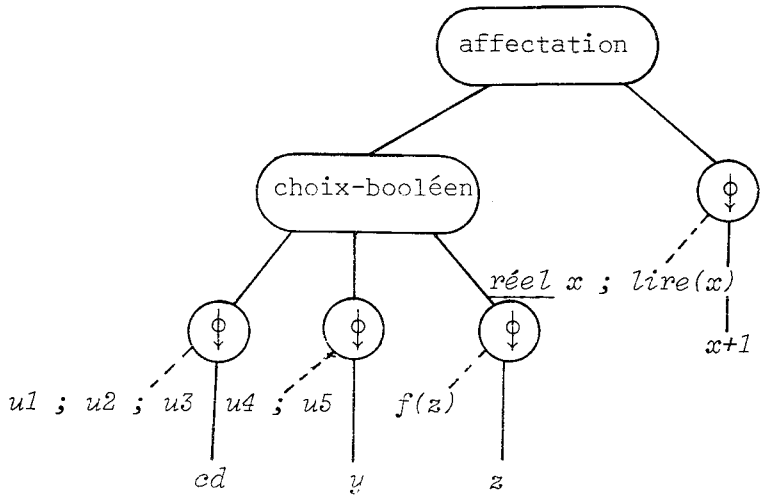
Les unités significatives n'ont pas changé.

Il en va différemment en ce qui concerne la génération de code. En effet, dans une proposition sérielle, les instructions s'exécutent séquentiellement. Il faut donc produire le code dans l'ordre des unités. Or, les premières unités n'apparaissent pas dans l'arbre réduit suffisant pour les modifications. L'arbre doit donc être tel qu'il ait les avantages de l'arbre réduit et qu'il permette l'accès aux unités non-significatives de la proposition sérielle. Pour cela le noeud (série) , qui a pour opérandes toutes les unités de la proposition sérielle, est remplacé par un noeud qui a deux opérandes : d'une part l'unité significative de la proposition sérielle, qui apparait effectivement dans cette représentation de l'arbre, et d'autre part une référence aux autres unités, qui vont rester sous forme postfixée jusqu'à la fin du traitement des modifications pour ce sous-arbre. Ce nouveau noeud est noté (↓) (descente) pour rappeler l'organisation par niveaux du texte postfixé. Il possède une décoration accès, qui reprèsente l'accès aux unités non-significatives qui sont sous forme postfixée.

L'arbre créé pour l'instruction (1) ci-dessus est le suivant :



Celui créé pour (2) est le suivant :



Lorsque le traitement des modifications a été réalisé sur l'arbre réduit, l'algorithme de génération de code prend en compte l'opérande gauche du noeud (↕) avant son opérande droit, de manière à respecter l'ordre des unités de la proposition sérielle.

2 - REPRESENTATION DE L'ARBRE ABSTRAIT -

L'arbre abstrait va donc exister en mémoire sous deux formes :

1) - postfixée - pour tout l'arbre
2) - avec pointeurs explicites - pour des parties d'arbre.

Sachant que l'arbre réduit doit permettre l'accès aux unités non-significatives qui n'y apparaissent pas, unités pour lesquelles des arbres réduits sont construits ensuite, il faut trouver une organisation adéquate du texte postfixé.

3 - SEGMENTATION DU TEXTE POSTFIXE -

La segmentation du texte postfixé obéit aux deux règles suivantes :

(1) Le texte postfixé est découpé en niveaux, tels qu'un arbre réduit se trouve sur un seul niveau, et les unités non-significatives au niveau immédiatement inférieur. Elles sont remplacées dans l'arbre par le noeud "descente" (↕)

(2) La séquence d'unités non-significatives d'une même proposition sérielle se trouve située sur un seul niveau, et se termine par une marque de "remontée" ↕.

notation : Afin de faciliter la lecture, les constructions Algol 68 postfixées sont représentées sous forme infixée soulignée.

Exemples : *c := a + b* pour *c a b + :=*

Exemples de segmentation du texte postfixé :

(1)- *début ent i, j ; lire(i) ; j := f(i) ; imprimer(j) ; j fin*

niveau (1) : *début* ↕ *j fin*

niveau (2) : *ent i, j ; lire(i) ; j := f(i) ; imprimer(j);* ↕

(2)- *si lire(i) ; i > 0 alors j := (i := -i ; imprimer(i) ; f(i)) ; j+1*
sinon g(i) fsi

niveau (1) *si* ↕ *i > 0 alors* ↕ *j+1 sinon g(i) fsi*

niveau (2) *lire(i)* ; ↕ *j := (*↕ *f(i)) ;* ↕

niveau (3) *i := -i ; imprimer(i) ;* ↕

Note : ↕ est un noeud de l'arbre tandis que ";" est un séparateur entre deux sous-arbres et "↕" une marque de remontée qui indique la fin d'une séquence d'unités non-significatives.

Nous montrons maintenant l'organisation des traitements réalisés sur l'arbre.

4 - IMBRICATION DES PHASES DE TRAITEMENT DES MODIFICATIONS ET DE GENERATION DE CODE -

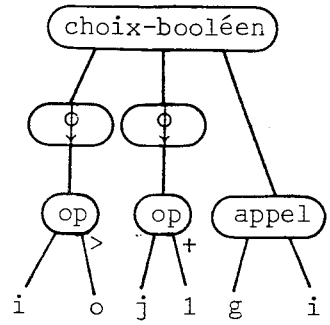
Le premier arbre qui est construit à partir du texte postfixé est l'arbre réduit du programme principal; il se trouve - seul - au niveau (1) . Le traitement des modifications est fait sur cet arbre en ne tenant pas compte des noeuds (↕) , et la génération de code commence. Comme elle doit produire du code pour toutes les unités, à chaque noeud (↕) , la séquence d'unités non-significatives qui se trouve au niveau inférieur est prise en compte et le triptyque - construction d'arbre, traitement des modifications, génération de code - est appliqué à chaque unité jusqu'à la rencontre de la marque "↕". Pour chacune de ces unités, le traitement peut être récursif, si elle contient un autre noeud (↕) Enfin, après remontée, la génération continue au niveau supérieur.

Nous illustrons ce schéma par son application à l'exemple (2) ci-dessus.

PRESENT en MEMOIR

Création arbre (1)
modifications (1)
début génération (1)

(1)



(1)

création arbre (2)
modifications (2)
génération (2)
destruction arbre (2)

(2) entrée-sortie .lire

i

(1) et (2)

(3) affectation

j — début — appel — f — i

(1) et (3)

création arbre (3)
modifications (3)
début génération (3)

(4) affectation

i — op — i

(1) (3) et (4)

création arbre (4)
modifications (4)
génération (4)
destruction arbre (4)

(5) entrée-sortie .imprimer

i

(1) (3) et (5)

création arbre (5)
modifications (5)
génération (5)
destruction arbre (5)

fin génération (3)
destruction arbre (3)

(1) et (3)

fin génération (1)
destruction arbre (1)

(1)

L'algorithme général est le suivant :

Algorithme général

Considérons une proposition sérielle constituée de n unités :

U1 ; U2 ; ... Ui ; ... Un

Le texte postfixé est produit à partir du niveau courant i

niveau i Φ Un

niveau i+1 U1 ; U2 ; ... Ui ; ... Un-1 ; Φ

niveau i+2 ...

Les unités non-significatives sont produites au niveau i+1 et si une de ces unités contient elle-même une proposition sérielle, ses unités non-significatives sont produites au niveau i+2, et ainsi de suite. Le nombre de niveaux correspond au degré de profondeur des propositions sérielles. Il est indépendant de la taille totale du programme. Il en est de même pour le nombre d'arbres présents à un instant donné en mémoire.

Appelons CMG - Construction, Modifications, Génération - le traitement appliqué à chacune des unités du texte postfixé :

CMG (Uj) : - Construction de l'arbre (Uj)
- Traitement des modifications (Uj)
- Génération de code (Uj)
- Destruction de l'arbre (Uj)

Le noeud Φ est pris en compte par la génération de code, qui peut être schématisée comme suit :

Génération de code (Uj) : si noeud courant = Φ
alors descendre au niveau inférieur (accès de Φ) ;
tant que noeud courant ≠ Φ
faire CMG (noeud courant) fait ;
Remonter au niveau supérieur
fsi ;
Génération effective (Uj)

La séquence d'appels pour une proposition sérielle de n unités est la suivante :

- Création de l'arbre (Un)
- Traitement des modifications (Un)
- Génération de code (Un)

  Descendre

  i de 1 à n-1 :
  - Création de l'arbre (Ui)
  - Traitement des modifications (Ui)
  - Génération de code (Ui) ⊸ Génération effective (Ui)
  - Destruction de l'arbre (Ui)

  Remonter

  Génération effective (Un)
- Destruction de l'arbre (Un)

L'appel initial étant : CMG (programme particulier)

Nous avons traité le problème de la segmentation lorsque la valeur d'une proposition sérielle est fournie par une seule unité, ce qui n'est pas toujours le cas. Voyons ce qui se passe lorsqu'une proposition sérielle contient plusieurs unités significatives.

Cas de plusieurs unités significatives

En Algol 68, comme dans d'autres langages de programmation, une proposition peut avoir plusieurs unités significatives (Return PL/1). L'unité qui précède un acheveur est significative, puisque l'acheveur (*exit*) marque une sortie de la proposition sérielle. Il peut y avoir plusieurs acheveurs dans une proposition sérielle, et toutes les unités significatives doivent apparaître dans l'arbre réduit. Elles sont reliées par le noeud "acheveur", et les unités non-significatives sont produites, comme précédemment, au niveau inférieur.

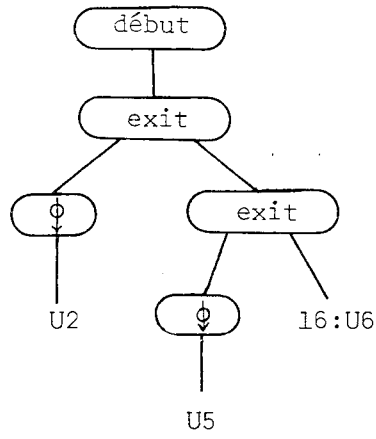
Exemple : *début U1 ; U2 exit l3 : U3 ; U4 ; U5 exit l6 : U6 fin*

texte postfixé :

niveau 1 : *début* ϕ *U2 exit* ϕ *U5 exit l6 : U6 fin*

niveau 2 : *U1 ;* ϕ↑ *l3 : U3 ; U4 ;* ϕ↑

arbre du niveau 1 :



Ordre de parcours de l'arbre

Le traitement des modifications et la génération de code peuvent conduire à un parcours de l'arbre qui ne correspond pas à l'ordre canonique des noeuds (Φ). Il est alors, nécessaire, pour la génération de code, de disposer au niveau du noeud (Φ), de l'accès aux unités non-significatives qui lui sont associées. Par contre, les unités non-significatives attachées à un noeud (Φ) sont traitées séquentiellement (traitement CMG).

5 - PRODUCTION D'UN TEXTE POSTFIXE SEGMENTE -

5.1 - Critères de segmentation -

Dans notre compilateur, c'est le deuxième passage qui produit l'image postfixée du texte source. Il faut donc savoir à quel moment changer de niveau. Cette question se pose dans les cas suivants :

- Avant la première unité d'une proposition sérielle , ou celle qui suit un acheveur , il faut décider si la production du texte continue au niveau courant ou s'il faut descendre d'un niveau.

- Avant toute autre unité, il faut savoir si elle est significative, auquel cas il faut remonter au niveau supérieur.

Exemple : *début* ↓? *U1* ; ↑? *U2* ; ↑? *U3* *exit* ↓? *l4:U4* ; ↑? *U5* *fin*

oui non oui oui oui

niveau 1 *début* Φ *U3* *exit* *U5* *fin*

niveau 2 *U1* ; *U2* ; ↑Φ *l4:U4*;↑Φ

La question de la descente se pose au début de chaque proposition sérielle et après chaque acheveur.

Critère : Descendre si la prochaine unité est suivie par ";". Dans les autres cas elle est significative (suivie par *exit*, *fin*, *alors*, ...).

La question de la remontée se pose à chaque symbole ";".

Critère : Remonter si la prochaine unité n'est pas suivie par ";". L'unité est significative et doit être produite au niveau supérieur.

5.2 - Rôle du premier passage -

Le premier passage produit un texte intermédiaire qui est une image enrichie du texte source. Les symboles où se pose la question du changement de niveau doivent être marqués :

- Délimiteurs de proposition sérielle :

  début, "(", *si*, *alors*, *sinon*, *cas*, *tant que*, *faire* ...

- ";"

- *exit*

A chacun de ces symboles est attribué un booléen qui indique si la prochaine unité est non-significative (NS) ou significative (S), ou, ce qui revient au même, suivie ou non par un ";".

Exemple :	*(*	*U1*	*;*	*U2*	*;*	*U3*	*exit*	*l4:U4*	*;*	*U5*	*)*
		NS		NS		S	NS		S		

Les critères de changement de niveau sont alors simples à mettre en oeuvre

- Symboles débutant une proposition sérielle : *début*, *si*, *(*, ... — Descendre si marqué Non-Significatif
- *exit*
- *;* — Remonter si marqué Significatif

Exemple :	*(*	*U1*	*;*	*U2*	*;*	*U3*	*exit*	*l4:U4*	*;*	*U5*	*)*
		NS		NS		S	NS		S		
		↓				↑	↓		↑		

Remarques : Les symboles "," ne jouent aucun rôle dans le marquage, car ils ne causent jamais de changement de niveau.

- Dans une proposition collatérale, une liste de paramètres ou une liste de bornes, toutes les unités sont significatives et restent au même niveau.
- Les virgules séparant des déclarations sont considérées comme des ";". A ce titre, elles sont marquées.
  Il faut remarquer qu'une déclaration n'est jamais significative et que toutes les déclarations d'une proposition sérielle sont produites au même niveau.

Algorithme de marquage

Le marquage d'une unité est reporté sur le symbole qui la précède. Or on ne connaît le type de l'unité, qu'à la lecture du symbole qui la suit. De plus, une unité peut contenir récursivement d'autres propositions sérielles. Une pile est utilisée. Les symboles à marquer sont repérés par leur numéro d'unité, qui est une valeur incrémentée sur de nombreux délimiteurs. L'algorithme de marquage est défini par les actions réalisées sur les symboles suivants :

1. *début* "(" *si* *cas* *tant que* *faire*
   - empiler le symbole courant

2. *fin* ")" *faire* {seulement si précédé de *tant que*} *fait*
   - marquer le symbole sommet de pile SIGNIFICATIF et le dépiler.

3. *alors* *dans* *sinon* *sinsi* *hors* "|" "|:"
   exit
   - marquer le symbole sommet de pile SIGNIFICATIF et le dépiler
   - empiler le symbole courant

4. ";"
   - marquer le symbole sommet de pile NON-SIGNIFICATIF et le dépiler.
   - empiler le symbole courant.

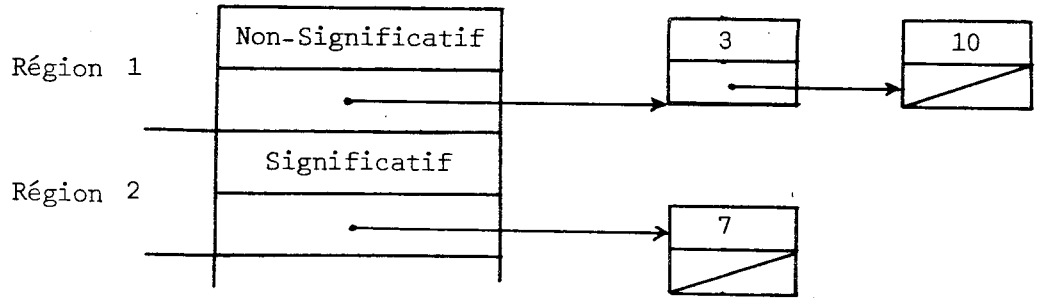
Transmission de la marque

Les symboles sont marqués au premier passage et utilisés au second pour la production du texte postfixé. La marque doit donc être enregistrée dans le texte intermédiaire. Celui-ci est constitué d'une image du texte source, qui est produite au fur et à mesure de sa lecture, et de tables. Comme on ne connaît la valeur de la marque qu'après avoir lu une fraction indéterminée du texte, elle est mise dans les tables par région, qui sont indexées par le numéro de bloc courant. Les symboles d'ouverture de bloc sont marqués directement dans la table, et les symboles précédant une unité significative sont chaînés depuis cette table.

Exemple :

```
                          ┌──── Région 2 ────┐
début U1 ; U2 ; U3 := (U4 exit L5:U5 ; U6 ; U7) exit L8:U8 ; U9 fin
  1      2    3       4    5         6    7   8   9         10    11
```

table des parenthèses



Le symbole *début* de la région 1 est marqué Non-Significatif car l'unité qu'il précède, U1, n'est pas significative. Dans la liste des unités significatives de cette région apparaissent les unités 3 et 10, désignées par le numéro du délimiteur qui les précède. Le symbole (de la région 2 est marqué significatif, car l'unité U4 est significative. Seule, l'unité U7 apparaît dans la liste des autres unités significatives de la proposition sérielle correspondante.

5.3 - Rôle du second passage -

Le texte postfixé est produit sur différents niveaux. Chaque niveau est constitué de "lignes" de taille fixe chaînées entre elles, repérées depuis un directoire. Nous donnons la description en Algol 68 des structures de données et des routines utilisées.

```
mode niveau = struct (rep niveau niveau supérieur,
                              niveau inférieur,
                      rep ligne première ligne,
                              deuxième ligne,
                      ent index courant co dans la ligne courante de ce
                                  niveau co) ;
mode ligne = struct ([1 : taille ligne] ent texte, co postfixé co
                      rep ligne ligne précédente,
                              ligne suivante) ;
rep ligne ligne courante  := nil ;
rep niveau niveau courant := nil ;
proc nouvelle ligne = rep ligne : co allocation d'une ligne au niveau
                                  courant co
                      début tas ligne L := (fant, nil, nil) ;
                            index courant de niveau courant := 1 ;
                            ligne précédente de L := ligne courante ;
                            L
                      fin ;
proc nouveau niveau = rep niveau :
                      début ligne courante := nil ;
                            tas niveau T := (niveau courant, nil,
                                             nouvelle ligne, nil,
                                             0)
                            ligne courante := dernière ligne de T
                                           := première ligne de T ;
                            T
                      fin ;
```

```
proc créer = (ent valeur) neutre:co production d'un élément de texte postfixé
                                    au niveau courant co
        début (texte de ligne courante) (index courant de niveau
              courant) := valeur ;
              si (index courant de niveau courant + := 1) > taille ligne
              alors ligne courante := ligne suivante de ligne courante
                                   := nouvelle ligne
              fsi
        fin ;
proc descendre = neutre : co descente au niveau inférieur co
        niveau courant := si rep rep niveau T = niveau inférieur
                                                de niveau courant ;
                             rep niveau (T) est nil
                          alors T := nouveau niveau
                          sinon ligne courante := dernière ligne de T ;
                                T
                          fsi ;
proc remonter = neutre : co remontée au niveau supérieur co
        (niveau courant := niveau supérieur de niveau courant ;
         ligne courante := dernière ligne de niveau courant) ;
proc initialisation = neutre : niveau courant := nouveau niveau ;
```

Ces procédures correspondent à des actions de compilation appelées sur les symboles *exit* et ";".

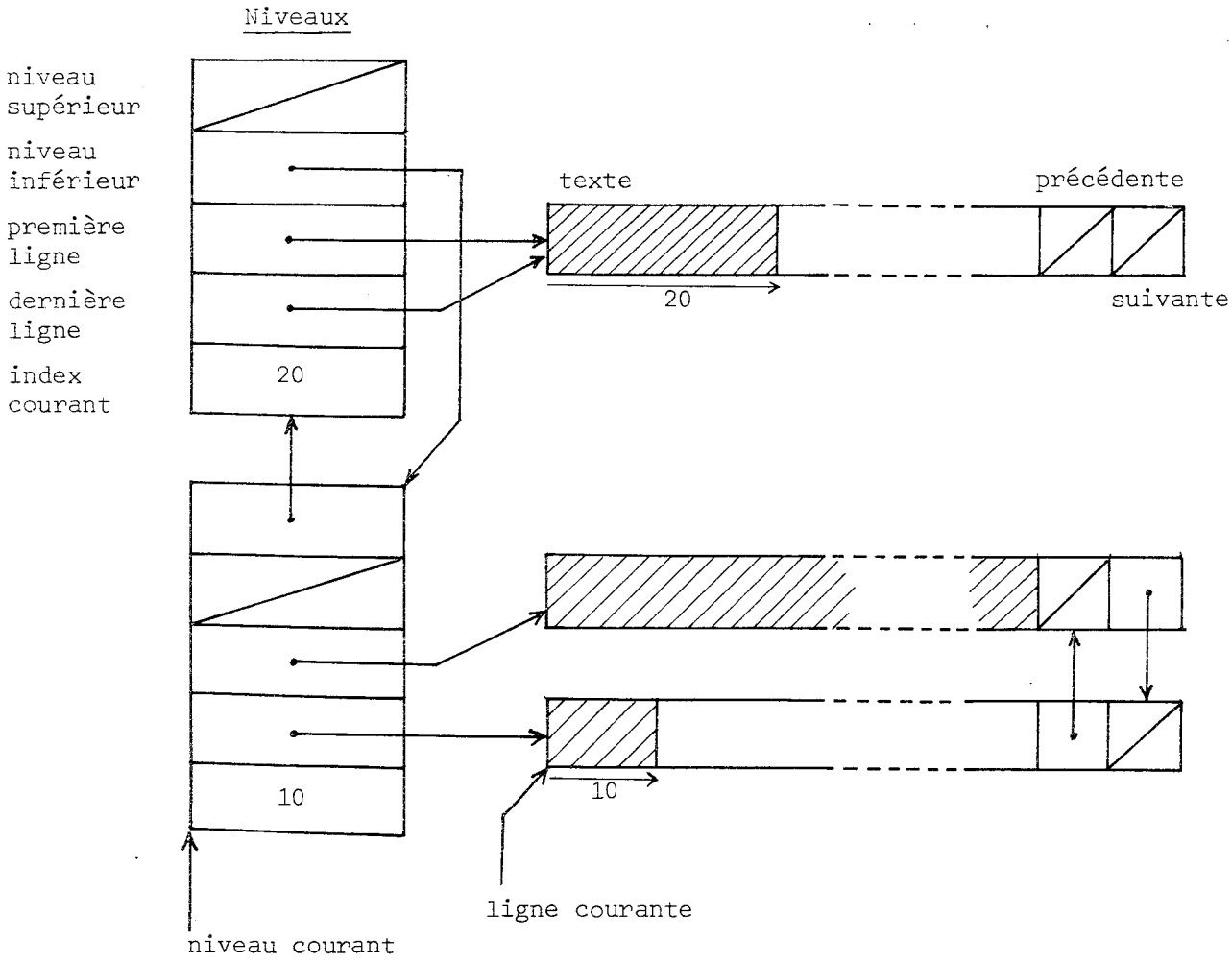
- *exit* et symboles débutant une proposition sérielle (*début*, "(", *si* ...) :

  si marque = Non-Significatif alors descendre fsi

- ";"

  si marque = significatif alors remonter fsi

La forme du texte produit est la suivante :



Les lignes sont des zones de taille fixe chaînées entre elles pour un même niveau. Le nombre de niveaux, aussi bien que le nombre de lignes par niveau, n'est pas limité . Cette organisation se prête en outre à l'utilisation de mémoires périphériques si nécessaire.

6 - CONCLUSION -

Les écrivains de compilateur qui ont choisi d'utiliser un arbre comme texte intermédiaire sont confrontés au problème de la taille de cet arbre. La méthode de segmentation que nous proposons permet de réduire considérablement la taille de l'espace mémoire requis pour représenter cet arbre. Elle est en accord avec le principe de localité, que nous avons essayé de respecter dans notre compilateur : on ne dispose pour le traitement d'un sous-arbre que des informations nécessaires à ce traitement. Nous donnons dans l'annexe 3 un exemple détaillé de segmentation d'un programme, et nous montrons l'imbrication des traitements associés.

III.4 - REPRESENTATION DES DECORATIONS - ALGORITHMES -

Dans ce chapitre nous nous intéressons à la représentation des principales décorations et aux algorithmes construisant ou utilisant ces représentations.

1 - REPRESENTATION DES DECORATIONS FONDAMENTALES -

Parmi l'ensemble des décorations fondamentales, nous distinguons deux catégories :

- les décorations de base,
- les décorations de modification.

1.1 - Décorations de base -

D'après leur signification, les décorations de base sont regroupées en trois classes.

- La décoration mode, composée :
  - de la représentation, à la compilation, du mode correspondant
  - des caractéristiques communes, pendant l'exécution, à toute valeur de ce mode.
    Exemple : taille de la partie statique de la représentation d'une valeur de ce mode.

- Les décorations identificateur, indicateur, opérateur, correspondant aux objets déclarés dans le programme.
  Chacune de ces décorations est formée :
  - de la déclaration mode associée au mode de l'objet déclaré
  - d'une fonction d'identification de l'objet déclaré
    Exemple : numéro lexicographique de l'identificateur
  - des informations statiques permettant de produire le code réalisant l'accès à la valeur possédée par l'objet déclaré, et sa manipulation.
    Exemple : déplacement dans la zone des variables (de l'environnement de déclaration) de l'emplacement réservé aux valeurs possédées par l'objet déclaré.

- La décoration bloc, associée à une région. Elle est formée :
  - de l'ensemble des décorations identificateur, indicateur et opérateur correspondant aux objets déclarés dans la région (sans tenir compte des régions internes).
  - des informations statiques concernant :
    . la position de cette région dans le programme,
    (Exemple : niveau statique d'imbrication en compte de régions),
    . l'ensemble des emplacements prévus, à l'exécution, (dans la zone des variables de l'environnement dont fait partie la région) pour les représentations des valeurs possédées par les objets déclarés dans cette région.
    Exemple : taille du fragment de la zone des variables correspondant aux déclarations de cette région.

Au vu de ces trois classes, on s'aperçoit qu'il y a une hiérarchie entre elles.

Le niveau le plus général est formé des décorations bloc, puis viennent les décorations identificateur, indicateur et opérateur, et enfin, au niveau le plus bas nous trouvons les décorations mode. En conséquence, nous avons utilisé une représentation arborescente dont nous détaillons les trois étages successifs.

1.1.1 - Représentation de la décoration mode -

La notion de mode dans un programme Algol 68 n'est pas directement liée à la structure de blocs du langage. Il est très fréquent de trouver le même mode utilisé à des endroits différents dans un même programme. C'est pourquoi nous regroupons les représentations de ces décorations dans une structure de données globale, c'est-à-dire accessible depuis tout point du programme pendant la phase de compilation. Cette structure est une table, appelée la table des classes, dont chaque élément correspond à un mode Algol 68. Pour des raisons de commodité d'utilisation de cette table (pour que ses éléments soient tous de la même taille), chaque élément ne contient pas la représentation du mode correspondant mais l'accès à cette représentation. Si nous prenons la précaution d'attribuer toujours le même élément à des modes égaux (but de l'algorithme d'équivalence de modes), pour qu'il n'y ait pas deux éléments qui représentent le même mode, nous avons une correspondance biunivoque entre les éléments de cette table et l'ensemble des modes distincts apparaissant dans le programme.

Nous pouvons alors attribuer à chaque mode un numéro caractéristique : l'index dans la table des classes de l'élément associé à ce mode.

Les représentations des divers modes sont rassemblées dans un tas appelé la table des modes. Chaque représentation est formée :

- d'un code identifiant le type du déclareur,
- du nombre de sous-modes constituants, lorsque celui-ci n'est pas induit par le type du déclareur,
- des numéros des sous-modes constituants.

Exemples :

rep ent est représenté par :
(code de *rep*, numéro de *ent*)

struct (réel x, rep ent y) est représenté par :
(code de *struct*, nombre de champs (2), numéro de *réel*, sélecteur *x*, numéro de *rep ent*, sélecteur *y*)

union (bool, compl) est représenté par :
(code de *union*, nombre de sous-modes (2), numéro de *bool*, numéro de *compl*)

Si nous considérons l'ensemble formé de la table des classes et de la table des modes, nous avons une structure de graphe dont les noeuds sont les enregistrements de la table des modes. L'accès à un noeud du graphe nécessite d'indexer la table des classes avec le numéro du mode correspondant à ce noeud.

1.1.2 - Représentation des décorations indicateur, identificateur et opérateur -

Contrairement aux précédentes, ces décorations sont liées à la structure de bloc du langage puisqu'elles correspondent aux déclarations faites dans chaque région du programme source. Les représentations de ces décorations vont être regroupées par région. A chaque région est associée une table appelée table des déclarations (en pratique, pour des raisons de commodité, il y en a quatre, correspondant aux déclarations de mode, de priorité, d'identificateurs et d'opérateurs) contenant les représentations de ces décorations. Dans chaque élément de cette table, la décoration mode est représentée par le numéro du mode de l'objet déclaré (ce numéro sert à indexer la table des classes).

En définitive, chaque objet déclaré dans une région est caractérisé par l'accès à la table de la région et par son index dans cette table. Si un même objet est déclaré dans plusieurs régions, il apparaît dans chacune des tables correspondantes. Bien entendu, chaque utilisation de cet objet ne se réfère qu'à une seule de ces tables. C'est le rôle des processus d'identification de décider du choix de cette table.

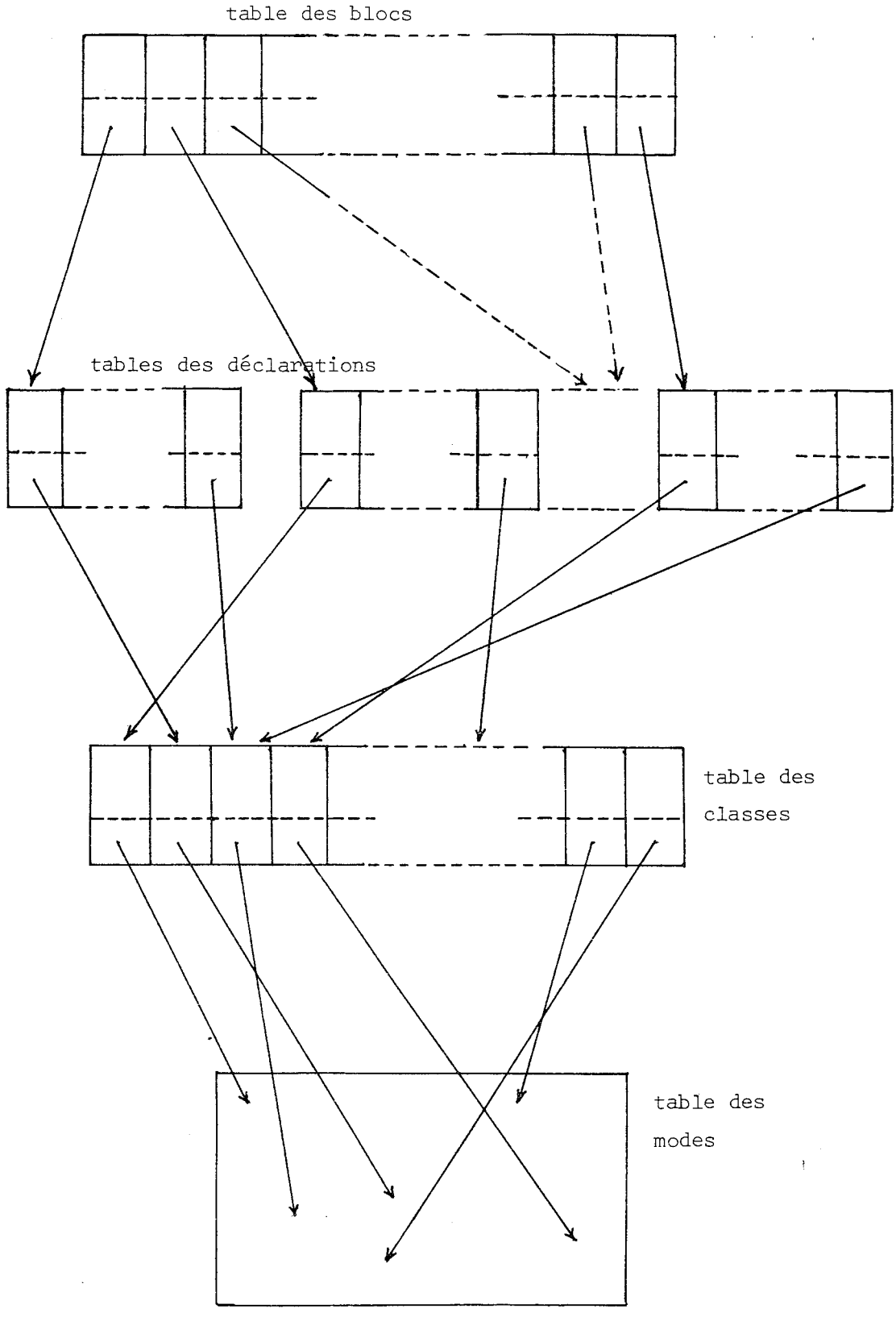
Les quatre tables associées à une région se nomment :

table des indicateurs de mode,
table des indicateurs de priorité,
table des identificateurs,
table des opérateurs.

1.1.3 - Représentation de la décoration bloc -

Ces décorations sont liées, bien sûr, à la structure de blocs du programme ; mais cette dépendance n'existe que vis à vis des décorations identificateur, indicateur et opérateur correspondant aux déclarations de chaque région. Pour des raisons de commodité d'utilisation, les décorations bloc sont regroupées dans une table globale appelée la table des blocs, c'est-à-dire accessibles depuis tout traitement réalisé pendant la phase de compilation. Chaque élément de cette table correspond à la décoration bloc associée à une région. En conséquence nous pouvons caractériser une région par l'index (dans cette table) de l'élément lui correspondant. Afin d'avoir une taille fixe pour les éléments de cette table, l'ensemble des décorations associées aux déclarations d'une région est représenté, dans chaque élément, par l'accès à la table des déclarations correspondante (il y en a quatre, en réalité), définie au paragraphe précédent.

1.1.4 - Schéma récapitulatif -



1.2 - Décorations de modification -

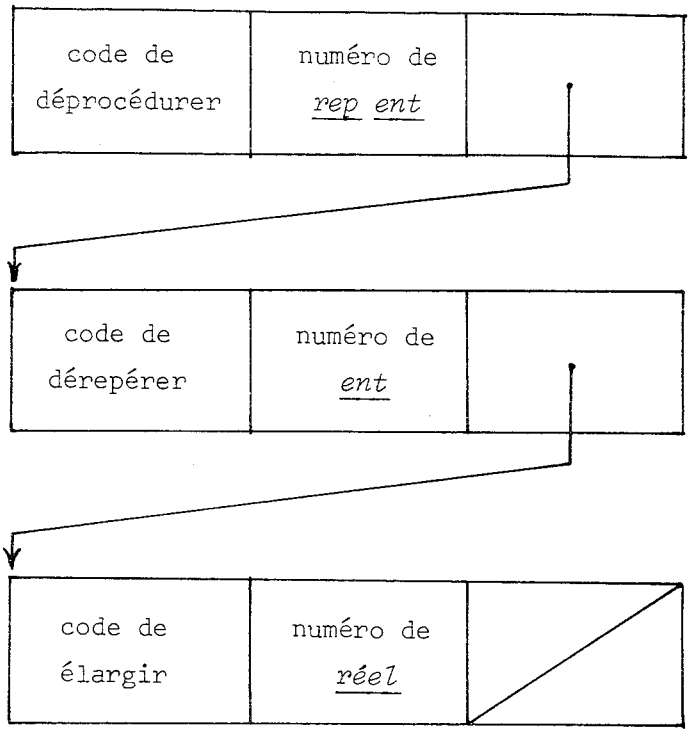
Ces décorations correspondent aux modifications que le langage permet d'appliquer à une valeur. Chaque valeur délivrée par une construction syntaxique possède un mode qui lui est propre. Ce mode n'est pas forcément celui requis pour la construction syntaxique. C'est le rôle de l'algorithme de création des modifications, de déterminer l'ensemble des modifications à appliquer successivement à une valeur pour obtenir une valeur du mode demandé.

Nous représentons une décoration modification par une liste, dont chaque élément correspond à une modification à appliquer à la valeur.

Un élément est formé :

- d'un code identifiant la modification.
  (Exemple : code de dérepérer
  code de élargir ... etc ...).
- du numéro du mode de la valeur obtenue par application de cette modification.
  (Exemple : si l'on applique une modification dérepérer à une valeur de mode *rep ent*, nous trouvons le numéro du mode *ent* dans l'enregistrement de cette modification)
- du pointeur vers l'enregistrement de la modification suivante, à appliquer à la valeur. Ce pointeur a la valeur *nil* pour la dernière modification de la liste.

Exemple : Soit une construction délivrant une valeur de mode *proc rep ent* alors que le mode demandé est *réel*. La décoration modification associée à cette valeur est représentée par :



1.3 - Conclusion -

Les représentations que nous avons adoptées pour les décorations n'ont rien d'original vis à vis de celles déjà utilisées dans les compilateurs des langages à structure de blocs. La notion de mode (qui est une généralisation de la notion de type apparaissant dans les langages classiques) nous a obligés à définir et donc à représenter les décorations mode et modification. La diversité des objets qu'il est possible de déclarer a, d'autre part, introduit les décorations indicateur et opérateur. En réalité, la solution, que nous avons adoptée, a consisté à généraliser et à étoffer les représentations de graphe habituellement utilisées.

2 - ALGORITHMES -

Nous allons maintenant nous intéresser aux trois principaux algorithmes utilisant et(ou) construisant les décorations précédentes :

- l'algorithme d'équivalence des modes,
- l'algorithme d'identification,
- l'algorithme de création des modifications.

2.1 - Algorithme d'équivalence des modes -

Du fait des "compatibilités" de modes qu'il est nécessaire de réaliser pendant la compilation d'un programme Algol 68, le compilateur doit être capable de savoir si deux modes sont ou non équivalents, lors du traitement de certaines constructions.

Exemple :

```
début
    ent i, j ; réel k
    ...
    i := j ;
    i := k ;
    ...
fin
```

L'affectation *i := j* est légale car les modes sont compatibles. Il n'en est pas de même de la deuxième (*i := k*) qui doit donner lieu à un message d'erreur. Dans les deux cas le compilateur a besoin de comparer les modes en présence. Plusieurs solutions à ce problème peuvent être envisagées. Le compilateur peut par exemple, décider de tester l'équivalence de deux modes uniquement lorsque cela est nécessaire (équivalence partielle) ou bien systématiquement pour tous les modes existants (équivalence totale).

2.1.1 - Choix d'une équivalence totale -

Dans notre implantation nous avons fait le choix suivant : tous les modes apparaissant dans un programme (y compris ceux de l'environnement standard) sont systématiquement comparés entre eux, dans le but de détecter tous les modes équivalents.

Ce choix est discutable et nous n'essayons pas de le justifier pleinement. Signalons simplement les avantages d'une telle décision.

Considérons un ensemble de modes équivalents que nous appelons une "classe d'équivalence". Pendant le processus de compilation, chaque élément de cet ensemble peut être caractérisé par un même numéro, celui (indiqué précédem-

ment) servant à l'indexation de la table des classes. Dans la table des modes, il n'est alors, plus nécessaire d'avoir les représentations de tous les modes de la classe, mais uniquement celle d'un "représentant". Pour une classe d'équivalence, il n'y a donc à créer qu'une seule représentation de mode et à calculer qu'une seule fois les caractéristiques des représentations (à l'exécution) des valeurs de chacun des modes de la classe (cf. représentation de la décoration mode).

Le temps de calcul utilisé par un tel algorithme d'équivalence n'est pas négligeable et s'il n'y avait pas d'autres avantages il ne serait pas raisonnable de l'utiliser. Son principal intérêt consiste à simplifier énormément l'algorithme de création des modifications. A tout moment ce dernier algorithme a besoin de savoir si les deux modes qu'il manipule (mode a priori et mode a posteriori) sont équivalents, pour décider de l'arrêt (si les modes sont équivalents ou "incompatibles") ou de la continuation de son traitement.

2.1.2 - Réalisation de l'algorithme d'équivalence -

De nombreux écrivains de compilateurs pour le langage Algol 68 se sont penchés sur le problème de l'équivalence des modes, ce qui a donné lieu à de nombreux algorithmes. Parmi tous ces algorithmes on peut distinguer deux tendances :

- les algorithmes de test "un contre un " [KOSTER-1]
- les algorithmes de classes d'équivalence [ZOSEL]

2.1.2.1 - Algorithme de test "un contre un " -

D'une manière générale le principe est le suivant : on dispose d'un ensemble de modes ayant déjà été testés (initialement cela correspond aux modes de l'environnement standard) et donc des représentants des classes existantes. Chaque nouveau mode est comparé successivement avec chacun de ces représentants jusqu'à :

- soit épuiser la liste de toutes les classes déjà existantes. Dans ce cas le mode n'est équivalent à aucun autre. Il faut alors créer une nouvelle classe et son représentant. Le numéro caractéristique du mode est celui associé à cette classe.
- soit trouver un représentant qui lui est équivalent. Il n'y a pas de classe à créer, le numéro caractéristique du mode étant celui de la classe ainsi trouvée.

Nous avons utilisé cette méthode pour le traitement de tous les déclareurs, excepté ceux apparaissant dans les déclarations de mode, et contenant des utilisations d'indicateurs de mode récursifs.

Exemples :

1) *mode m1 = struct(rep ent a, rep m1 b) ;*

Dans cet exemple, seuls les déclareurs *rep ent* et *ent* sont traités de cette manière. Les déclareurs *rep m1* et *struct(rep ent a, rep m1 b)* correspondent à des modes récursifs.

2) *mode m1 = ... ;*
rep m1 x = ... ; rep ent y = ... ;

Le déclareur *rep m1* va être traité de cette manière car il n'appartient pas à une déclaration de mode. Ceci s'explique par le fait que connaissant le numéro du mode de *m1*, le déclareur *rep m1* est aussi simple à traiter que le déclareur *rep ent*.

Moyennant cette restriction quant à son utilisation, nous avons pu spécialiser cet algorithme de la manière suivante : le traitement d'un mode n'est réalisé qu'une fois terminé celui de ses sous-modes constituants. Rappelons que l'algorithme d'équivalence fournit pour chaque mode son numéro caractéristique. Dans ces conditions, la comparaison de deux modes se résume à comparer les numéros des sous-modes les constituant , après s'être assuré que ces deux modes sont du même type (deux déclareurs de repère, deux déclareurs de procédure avec le même nombre de paramètres, etc...).

Exemple : soit le mode *proc (rep réel, ent) réel*

Le traitement de ce mode n'est entrepris que lorsque celui de ses sous-modes *rep réel*, *ent*, *réel* est terminé. Nous disposons alors d'un mode "procédure à deux paramètres dont les numéros de mode sont respectivement : numéro de *rep réel* et numéro de *ent*, et délivrant un résultat de numéro de mode : numéro de *réel*". Ce mode est à comparer avec ceux (déjà existants) des procédures à deux paramètres. Cette comparaison se réduit à celle des numéros des sous-modes constituants (dans leur ordre d'apparition). Du fait de sa simplicité, cet algorithme est très intéressant. Malheureusement, ainsi spécialisé, il ne convient pas à certains modes apparaissant dans les déclarations de modes récursifs.

Exemple : *mode m1 = struct (rep m1 a, ent b)*

Le traitement du mode correspondant à *m1* nécessite d'avoir terminé celui du déclareur *rep m1* qui lui même demande d'avoir préalablement fait celui de *m1* ; ce qui est incompatible.

2.1.2.2 - Algorithme de classes d'équivalence -

Cet algorithme teste globalement tous les modes entre eux. Ce n'est qu'une fois l'algorithme terminé, que l'on dispose des classes d'équivalence. Le principe est le suivant :

- Il y a d'abord construction des classes initiales
  - les modes de base n'étant pas équivalents entre eux, chacun constitue le représentant d'une classe. Nous avons donc autant de classes qu'il y a de modes de base dans le langage.
  - on construit une nouvelle classe contenant tous les autres modes, c'est-à-dire tous les modes à tester (différents des modes de base).

- Les éléments de chaque classe sont supposés "momentanément équivalents". A tout instant, chaque mode existant a comme numéro celui de la classe dont il fait partie. Considérons l'ensemble ordonné des classes ne correspondant pas aux modes de base, que nous savons uniques et non équivalents (cet ensemble est primitivement formé d'une seule classe). L'algorithme opère par balayages successifs de cet ensemble, de la manière suivante :

Pas 0 : Sélectionner la première classe et exécuter le Pas 1.

Pas 1 : Sélectionner un élément de la classe en traitement. Cet élément est comparé avec tous les autres éléments de la classe. Cette comparaison est réalisée comme dans le cas de l'algorithme de test un contre un, c'est-à-dire que l'on compare les sous-modes des modes de même type. Si tous les éléments sont (momentanément) équivalents à l'élément choisi, on exécute le Pas 2 sinon on exécute le Pas 4.

Pas 2 : Si la classe en traitement est la dernière classe de l'ensemble ordonné exécuter le Pas 3 sinon sélectionner la classe suivante et exécuter le Pas 1.

Pas 3 : Si pendant le balayage (qui se termine) de l'ensemble des classes il y a eu création d'une ou plusieurs classes, c'est-à-dire s'il y a eu augmentation du nombre d'éléments de l'ensemble des classes, exécuter le Pas 0 sinon terminer l'algorithme.

Pas 4 : Créer une nouvelle classe formée de l'élément sélectionné, et de tous les éléments qui lui sont momentanément équivalents. Cette classe est rajoutée en fin de l'ensemble ordonné des classes existantes (elle est donc traitée à la fin du balayage courant). Les éléments mis dans la nouvelle classe sont retirés de la classe courante. L'algorithme se poursuit par l'exécution du Pas 1.

L'algorithme se termine donc, lorsqu'aucune classe n'a été créée au cours d'un balayage de l'ensemble des classes. En effet, il est inutile d'entreprendre un autre balayage qui se déroulerait dans les mêmes conditions que le précédent et qui, de ce fait, ne fournirait aucune nouvelle classe.

Cet algorithme n'est pas très performant. Il peut être amélioré grâce à la remarque suivante :

En cours d'algorithme, lorsqu'une classe se réduit à un seul élément, cet élément ne peut être équivalent à aucun autre.

En conséquence, cette classe, qui se comporte comme celles des modes de base, peut être retirée de l'ensemble des classes à balayer, ce qui diminue le nombre d'éléments de cet ensemble.

Une deuxième amélioration peut être apportée en augmentant le nombre de classes initiales. Au lieu de mettre tous les modes dans la même classe on peut les répartir dans plusieurs, en utilisant des critères empiriques mais sûrs. Bien évidemment, il faut qu'aucun mode d'une des classes ainsi obtenues ne soit équivalent à un mode d'une autre classe.

Exemple : un mode *repère* et un mode *tableau* ne peuvent jamais être équivalents ni un mode *procédure à deux paramètres* et un mode *procédure à trois paramètres*

Malgré toutes les améliorations possibles cet algorithme reste plus coûteux que le précédent. Son avantage est qu'il permet de traiter aussi bien les modes récursifs, apparaissant dans les déclarations de mode, que les autres. En effet, pour traiter un mode, il n'est pas nécessaire d'avoir déjà traité

ses sous-modes constituants. C'est pourquoi nous utilisons cet algorithme, mais en limitant son emploi aux modes récursifs apparaissant dans les déclarations de mode.

2.1.2.3 - Utilisation de ces algorithmes -

Dans notre compilateur, une première phase d'équivalence est entreprise à la fin du premier passage, lorsque nous avons collecté les déclareurs des indicateurs de mode déclarés dans le programme.

A cet instant, nous reprenons l'ensemble des déclareurs ainsi trouvés. Nous considérons tous les modes apparaissant dans chaque déclareur, en commençant par les sous-modes les plus "internes". Deux cas peuvent se produire :

- le mode n'est pas récursif. Dans ce cas, en utilisant l'algorithme de test un contre un, nous comparons ce mode avec tous les modes déjà traités.
- le mode est récursif. Nous ne lui faisons subir momentanément aucun traitement.

Ceci terminé, tous les modes non récursifs sont testés. Nous reprenons alors, tous les modes récursifs et les répartissons dans un certain nombre de classes. L'algorithme de classes d'équivalence est alors mis en oeuvre sur ces seuls modes qui ne peuvent en aucun cas être équivalents à ceux déjà traités (modes non récursifs).

Au cours du deuxième passage (lorsque le compilateur rencontre un générateur, une déclaration de variable ou d'identité, un forceur ou une déclaration d'opérateur) ou à certains moments de la phase de détection des modifications (lorsqu'un symbole *rep* est propagé à l'intérieur d'une structure ou sur les éléments d'un tableau par une opération de sélection ou d'indexation, par exemple ; on trouve la liste de ces cas en consultant le Rapport) il est nécessaire de comparer le mode trouvé avec tous les modes existants. Les problèmes posés par les modes récursifs des indicateurs de mode étant résolus, nous utilisons, pour faire l'équivalence, l'algorithme de test un contre un déjà cité.

2.2 - Algorithme d'identification -

2.2.1 - L'identification en Algol 68 -

Les caractéristiques des langages à structure de bloc sont très importantes au point de vue des déclarations. Dans les langages courants (PL/1,Algol W, ...) il est interdit de déclarer deux fois le même identificateur dans le même bloc (condition d'unicité de déclaration). Par contre si nous déclarons dans un bloc le même identificateur (lexicographiquement parlant) que dans un bloc englobant, la déclaration du bloc englobant n'est plus accessible pendant toute la durée de vie du bloc interne.

Exemple :

```
début
    ent x :
    ...
    début
        réel x ;
    ...
    fin ;
    ...
fin
```

Dans le langage Algol 68 nous retrouvons cette particularité :

- pour les déclarations d'identificateurs
- pour les déclarations d'indicateurs (déclarations de mode et de priorité)

Par contre, les déclarations d'opérateurs ne sont pas soumises aux mêmes règles, puisqu'il est possible de déclarer plusieurs opérateurs de même nom dans la même région en respectant certaines règles de "compatibilité". L'enregistrement des déclarations de mode et de priorité étant faite au premier passage, le compilateur réalise l'identification des indicateurs au deuxième passage. Il en est de même pour les identificateurs avec le traitement des déclarations au deuxième passage et la phase d'identification lors d'un parcours de l'arbre abstrait. Le même processus d'identification est utilisé dans les deux cas.

2.2.2 - But de l'algorithme -

Etant donné que le compilateur dispose, en temps voulu, des tables des déclarations par région, nous aurions pu adopter le processus classique suivant :

1) - A chaque ouverture de région, empiler les tables (ou leurs adresses) des déclarations de la région.

2) - A chaque occurrence d'utilisation, explorer la pile, en commençant par la tête, jusqu'à trouver la déclaration correspondante.

3) - A chaque fermeture de région, dépiler les tables (ou leurs adresses) des déclarations de la région.

Partant de la constatation que, dans un programme, il existe, en général, plus d'occurrences d'utilisations que de déclarations, il nous a semblé intéressant de mettre en jeu un algorithme nécessitant un temps très court pour établir le lien utilisation-déclaration, même si cela conduit à un travail supplémentaire sur les ouvertures et fermetures de région.

Le processus proposé permet, sur toute utilisation en un point quelconque d'un programme, d'identifier directement la déclaration correspondante.

2.2.3 - Processus d'identification -

Expliquons le processus sur l'identification réalisée pour les identificateurs, pendant un parcours de décoration de l'arbre abstrait. Pour disposer, depuis n'importe quel point de l'arbre, de l'ensemble des déclarations d'identificateurs accessibles, nous utilisons une table, appelée la table des dernières occurrences, ayant autant d'éléments qu'il existe d'identificateurs lexicographiquement différents. Chaque élément de cette table est de longueur fixe, et contient à tout instant, l'accès à l'enregistrement associé à la déclaration valide de l'identificateur correspondant. Cet accès est formé du numéro de la région contenant la déclaration accessible et du rang de l'enregistrement associé dans la table des identificateurs de la région. Les champs de cette table sont primitivement initialisés à zéro, sauf ceux associés aux identificateurs standard qui contiennent l'accès aux enregistrements correspondants.

On accède à un élément de la table des dernières occurrences en l'indexant avec le numéro de l'identificateur cherché. Examinons les traitements à réaliser au début et à la fin de chaque région.

- Traitement en début de région -

La table des dernières occurrences est à mettre à jour si la région contient des déclarations d'identificateurs. En parcourant la table des identificateurs de la région on obtient les numéros des identificateurs déclarés, et donc les indices des éléments à mettre à jour dans la table des dernières occurrences. Avant de modifier ces éléments il faut sauvegarder les valeurs qui s'y trouvent afin de pouvoir, en fin de région, restituer la table des dernières occurrences dans l'état où elle était avant l'entrée dans cette région. A cet effet, dans chaque enregistrement des tables d'identificateurs, est prévue une zone pour sauvegarder l'accès à l'enregistrement associé à la précédente déclaration du même identificateur.

- Traitement en fin de région -

D'après ce que nous venons de dire, à la fin d'une région, il faut restaurer l'état qu'avait la table des dernières occurrences avant le début de la région. Le parcours de la table des identificateurs de la région nous fournit non seulement les indices des éléments à modifier, mais aussi, pour chaque index, la valeur à restaurer. Il n'y a donc aucune difficulté à restaurer l'état initial de la table des dernières occurrences.

Avec une telle organisation, sur chaque utilisation d'un identificateur, l'accès à l'enregistrement correspondant à la déclaration est fourni simplement par l'indexation de la table des dernières occurrences, avec le numéro lexicographique de l'identificateur.

2.2.4 - Généralisation de l'algorithme -

Nous avons généralisé cet algorithme au cas des opérateurs. Etant donné que nous pouvons avoir plusieurs déclarations du même opérateur dans la même région, nous ne pouvons pas ranger tous les accès à leurs déclarations, dans la table des dernières occurrences, car elle ne dispose que d'une entrée par opérateur lexicographiquement différent.

Si nous analysons l'algorithme pour les identificateurs, nous remarquons que nous pouvons accéder, non seulement à la dernière déclaration, mais aussi à l'ensemble des déclarations de cet identificateur. En effet, dans l'enregistrement associé à la dernière déclaration, nous avons sauvegardé l'accès à l'enregistrement de la précédente déclaration de ce même identificateur. Depuis chaque entrée de la table, nous avons donc une chaîne reliant toutes les déclarations de l'identificateur correspondant.

Cette caractéristique va nous permettre d'utiliser l'algorithme, pour les opérateurs, de la manière suivante :

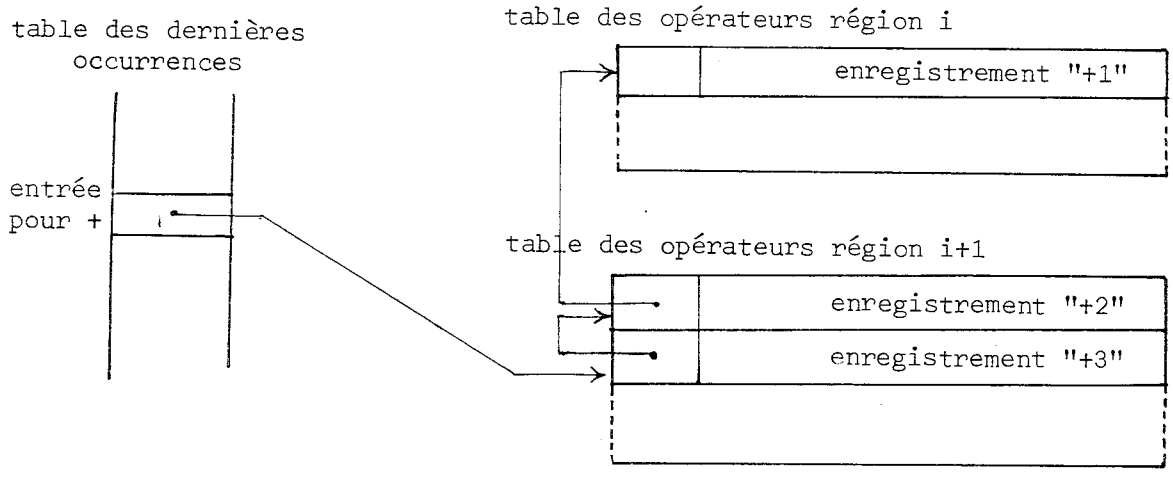
2.2.4.1 - A chaque entrée ou sortie de région, la table des dernières occurrences est mise à jour, comme indiqué pour les identificateurs.

Remarque : Pour restaurer en fin d'une région l'état initial de la table des dernières occurrences, la table des opérateurs doit être parcourue dans l'ordre inverse de celui utilisé lors de l'entrée dans la région.

Exemple : Soit le programme :

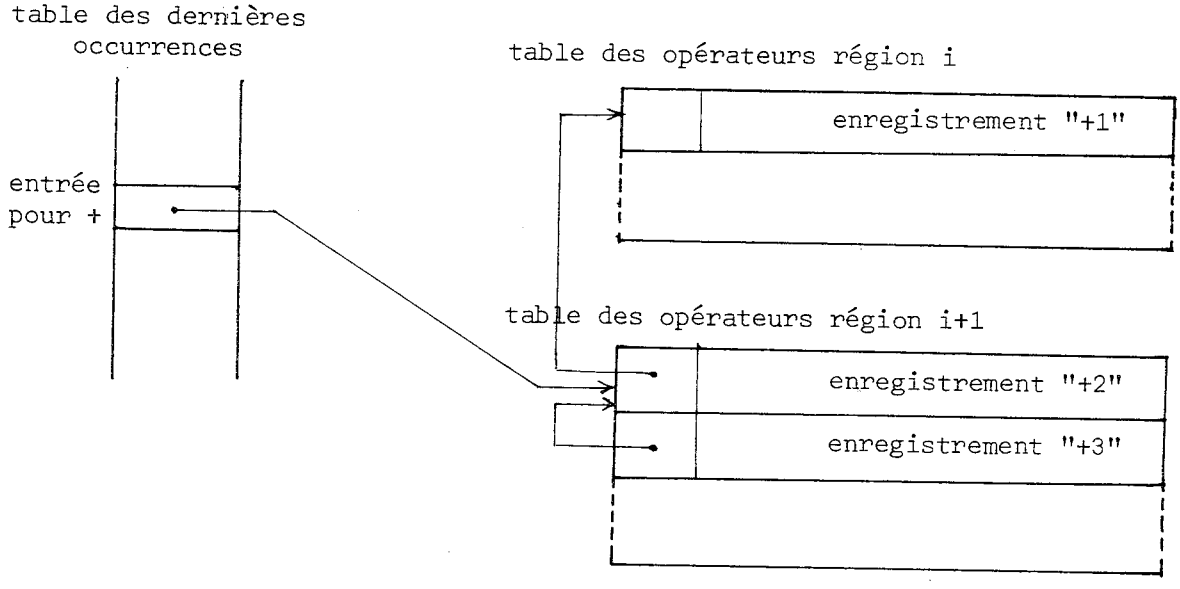
```
début co région i co
    op + = ... ; co opérateur +1 co
(A) début co région i + 1 co
        op + = ... ; co opérateur +2 co
        op + = ... ; co opérateur +3 co
        ...
(B) fin
    ...
fin
```

Au point (A), nous mettons à jour la table des dernières occurrences et nous avons la configuration suivante :



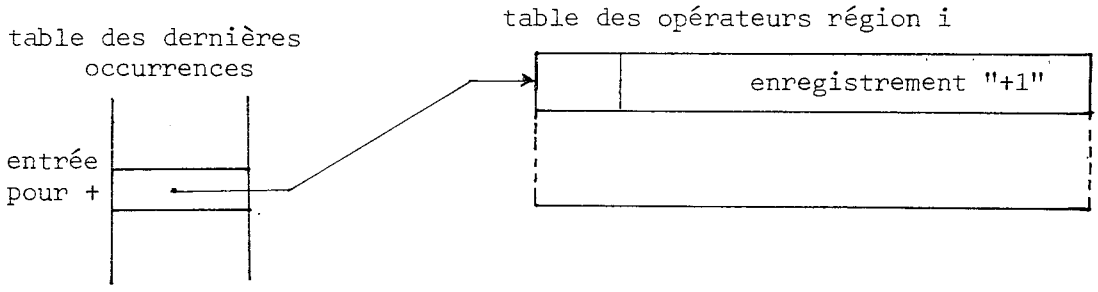
Au point (B) , la table des dernières occurrences est restaurée ; on peut schématiser son état après une restauration effectuée :

1) - par le même parcours séquentiel de la table des opérateurs que celui utilisé en (A) .



Remarquons alors que dans la région i, nous avons accès à des déclarations de la région i+1.

2) - par un parcours séquentiel, en ordre inverse, de la table des opérateurs.



C'est bien la configuration que nous avions en arrivant au point (A) .

2.2.4.2 - A chaque utilisation d'opérateur, la table nous donne l'accès à la première déclaration valide. Si elle ne convient pas, nous essayons les déclarations précédentes en explorant la chaîne construite pour cet opérateur, jusqu'à trouver une déclaration qui convienne, ou jusqu'à atteindre la fin de la chaîne. Nous retrouvons le principe de l'algorithme classique d'identification, quoique nettement amélioré grâce aux chaînages qui permettent de ne tester que les opérateurs ayant le même nom que celui utilisé.

L'identification des opérateurs étant réalisée pendant le même parcours de l'arbre que celle des identificateurs, nous utilisons une seule table des dernières occurrences. De ce fait, étant donné qu'une déclaration ou une utilisation d'opérateur est aussi une utilisation d'indicateur (de priorité), l'analyseur lexicographique du premier passage délivre des numéros différents pour un indicateur et un identificateur dont les noms sont formés de la même suite lexicographique. La même table des dernières occurrences peut ainsi servir à l'identification des indicateurs au deuxième passage.

2.2.5 - Evaluation -

2.2.5.1 - Etude comparative -

Cherchons à comparer en simplifiant beaucoup, la méthode classique et celle proposée. Pour cela, intéressons nous au cas des identificateurs, nous allons essayer de dénombrer le nombre d'accès aux tables d'identificateurs.

Soit un programme ayant N régions imbriquées numérotées de 1 à N. Nous supposons que dans la région de numéro k il y a D_k déclarations et U_k utilisations d'identificateurs.

- Méthode proposée -

L'accès aux tables d'identificateurs n'a lieu qu'en entrée et en sortie de région. Pour la région de numéro k nous avons un même nombre d'accès à la table des identificateurs : D_k, en début de région, et D_k, en fin de région. Le nombre total d'accès aux tables d'identificateurs pour le programme donné est :

$$C_p = 2 \sum_{k=1}^{N} D_k$$

Pour chaque utilisation aucun accès aux tables n'étant nécessaire, le coût de l'opération est négligeable.

- Méthode classique -

Pour évaluer le coût pour cette méthode, nous devons faire des hypothèses, notamment :

- sur la fréquence et,
- sur la localisation

des identificateurs, en fonction de la région de leurs déclarations.

Pour simplifier nous supposons l'hypothèse suivante :

Etant donné, dans la région de numéro k, une utilisation d 'identificateur, la probabilité que nous avons, de trouver la déclaration correspondante dans la région de numéro $j \leqslant k$ est :

$$\frac{D_j}{\sum_{i=1}^{k} D_i}$$

Cette hypothèse ne correspond pas très bien à la réalité car, généralement, les identificateurs les plus utilisés dans une région précise d'un programme donné, sont les identificateurs déclarés dans cette région ou dans la région la plus englobante. Cette distorsion ponctuelle s'atténue, en moyenne, lorsque l'on s'intéresse à un programme complet.

Avec cette hypothèse le coût total des identifications dans un programme peut s'écrire :

$$C_c = \sum_{k=1}^{N} (\text{coût de l'ensemble des utilisations } U_k) = \sum_{k=1}^{N} C_{c_k}$$

Evaluons C_{c_k}. Nous obtenons :

$$C_{c_1} = \frac{D_1 + 1}{2} U_1$$

$$C_{c_2} = \frac{D_1 + D_2 + 1}{2} U_2$$

...

$$C_{c_k} = \frac{D_1 + D_2 + \ldots + D_k + 1}{2} U_k$$

...

Nous en déduisons :

$$C_c = \frac{1}{2} \{D_1 \sum_{k=1}^{N} U_k + \ldots + D_i \sum_{k=i}^{N} U_i + \ldots + D_N U_N\} + \frac{1}{2} \sum_{k=1}^{N} U_k$$

- Comparaison des deux méthodes -

Elle est faite dans les conditions suivantes :

- tous les D_k sont sensiblement égaux à une valeur α
- tous les U_k sont sensiblement égaux à une valeur β

Dans ce cas précis, nous obtenons :

$$C_p = 2 \alpha \sum_{k=1}^{N} 1 = 2 N \alpha$$

$$\text{et } C_c = \frac{1}{2} = \alpha \beta \left[\sum_{k=1}^{N} + \sum_{k=2}^{N} + \ldots + 1 \right] + \frac{\beta}{2} \sum_{k=1}^{N} 1$$

$$= \frac{\alpha \beta N(N+1)}{4} + \frac{N\beta}{2}$$

En définitive nous avons :

$$C_p = \sigma(N\alpha)$$

$$C_c = \sigma(\alpha\ \beta\ N^2)$$

$\sigma(x)$ signifie de l'ordre de x.
(notation de P. Bachmann - Analytische Zahlentheorie - 1892)

Remarquons que le coût de la solution proposée ne dépend pas du nombre d'utilisations d'identificateurs.

2.2.5.2 - Avantages et inconvénients de la méthode -

Nous pouvons remarquer qu'il est très facile de vérifier la condition d'unicité de déclaration pour les identificateurs et les indicateurs (les opérateurs ne vérifient pas cette condition en Algol 68). Dans le traitement en entrée de région, il suffit de rajouter, pour chaque élément à modifier dans la table des dernières occurrences, un test pour savoir si cet élément donne déjà accès à l'enregistrement associé à un identificateur (ou à un indicateur) de la région (test du numéro de la région). Cela évite, lors du traitement d'une déclaration, d'avoir à la comparer avec toutes les déclarations déjà traitées pour la région.

Le coût en place mémoire de cette méthode en est le principal inconvénient. Pour utiliser cet algorithme il faut disposer de :

- la table des dernières occurrences dont chaque entrée contient l'équivalent d'un pointeur
- le champ de sauvegarde de ces pointeurs dans chaque enregistrement des tables par région.

Avec les conventions précédentes nous pouvons calculer la perte de place occasionnée par cette méthode.

1) - Au maximum, la table des dernières occurrences a autant d'entrées qu'il y a de déclarations. D'où sa taille :

$$T_{TDO} = \sigma\ (\alpha\ N)$$

2) - Nous avons autant de champs de sauvegarde qu'il y a de déclarations. D'où la taille de l'ensemble des champs :

$$T_{CS} = \sigma\ (\alpha\ N)$$

Nous en déduisons la taille totale nécessaire :

$$T_{TOT} = \sigma\ (\alpha\ N)$$

Remarquons que le coût supplémentaire, en espace mémoire, est, une fois de plus, indépendant du nombre d'utilisations.

2.3 - Algorithme de création des modifications -

2.3.1 - Introduction -

Rappelons que le langage Algol 68 a généralisé et clarifié la notion de modification qui n'apparait que très partiellement dans les autres langages (exemple : conversion de type "entier" vers le type "réel").

Il a été défini six modifications correspondant chacune à une action précise :

- déprocédurer — Ex : *proc réel* → *réel*
- dérepérer — Ex : *rep réel* → *réel*
- unir — Ex : *réel* → *union (réel, ent)*
  union (réel, ent) → *union (réel, ent, bool)*
- mettre en rang — Ex : *réel* → [ ] *réel*
  [ ] *réel* → [,] *réel*
  [,] *réel* → [ ] [,] *réel*
- élargir — Ex : *ent* → *réel*
  réel → *compl*
- neutraliser — Ex : *réel* → *neutre*

Le rapport de définition du langage donne les règles de composition possible de ces modifications.

Ainsi, cinq forces de contexte sont introduites selon la position syntaxique de l'objet à traiter :

- le contexte fort où toutes les modifications sont autorisées,
- le contexte ferme où seules les trois premières le sont,
- le contexte déterminant et le contexte faible où seules les deux premières le sont,
- le contexte mou, où seule la première est possible.

De plus, des directives existent sur l'ordre dans lequel ces modifications peuvent être appliquées à un mode.

La position syntaxique de l'objet permet de connaître la force du contexte, donc les modifications applicables, et fournit des informations sur le mode désiré. Deux cas peuvent se présenter :

- le mode désiré est entièrement déterminé par les règles syntaxiques ou des processus sémantiques. C'est le cas des contextes fort ferme et en partie du contexte déterminant.
- le mode désiré n'est connu que par son préfixe, c'est-à-dire par le (ou les) premier(s) symbole(s) le composant. C'est le cas pour le contexte faible, le contexte mou et, en partie, pour le contexte déterminant. Bien que limités, ces renseignements sont suffisants.

En définitive nous avons :

un mode de départ : mode a priori (mode propre),
un contexte (fort, ferme, déterminant, faible, mou),
et un mode requis : mode a posteriori (mode contextuel), complet ou pas.

Le problème est de déterminer la liste des modifications à appliquer au mode a priori, en respectant le contexte.

Considérons les constructions Algol 68 suivantes :

Ex : 1) *si* *b* *alors* ...

La partie de l'instruction comprise entre les symbole *si* et *alors* est dans un contexte déterminant et doit être de mode *bool* (ceci est indiqué dans la règle syntaxique correspondante).

Nous avons donc les identités :

mode a priori = mode de *b*,
contexte = déterminant,
mode a posteriori = *bool* .

Si le mode de *b* est *rep proc bool* la liste de modifications à appliquer est : dérepérer, déprocédurer.

2) *x* := ...

La partie gauche de l'affectation est dans un contexte mou et de mode commençant par le symbole *rep* (informations fournies par la syntaxe).

Nous avons donc les identités :

mode a priori = mode de *x*,
contexte = mou,
et le mode a posteriori doit commencer par *rep*.

Si le mode de *x* est *proc proc rep réel*, la liste des modifications à appliquer est : déprocédurer , déprocédurer . Il faut ensuite tester le mode obtenu pour savoir s'il commence bien par le symbole *rep* puis, si c'est le cas, compléter le mode a posteriori avec le mode obtenu (dans le cas présent, le mode a posteriori devient *rep réel*).

2.3.2 - Implantation d'un algorithme -

Dans ce paragraphe nous ne détaillons pas l'algorithme non récursif que nous avons utilisé. Pour cela nous renvoyons le lecteur à [68GRE-4]. Indiquons simplement les deux principales chaînes de traitement de cet algorithme.

Deux cas se présentent :

1) - Lorsque le mode a posteriori est entièrement connu, le problème consiste à déterminer la liste des modifications à appliquer au mode a priori pour obtenir le mode a posteriori, en respectant les directives fournies par le contexte.

Le cheminement peut s'effectuer :

a) - du mode a posteriori vers le mode a priori. Une fois la liste des modifications déterminée, le passage du mode a priori vers le mode a posteriori est possible par application, dans l'ordre inverse, de la même liste de modifications.

b) - du mode a priori vers le mode a posteriori. C'est en partie le cas pour le contexte déterminant.

c) - dans les deux sens. C'est le cas pour les contextes ferme et fort.

2) - Lorsque le mode a posteriori est incomplet, il faut appliquer au mode a priori toutes les modifications possibles et vérifier que le mode obtenu est en accord avec les renseignements (fournis par la syntaxe) concernant le mode a posteriori. Lorsque ceci est vrai, il reste à compléter le mode a posteriori en utilisant le mode obtenu.

2.3.3 - Conclusion -

L'ensemble de ces algorithmes constitue une implantation possible de la détection des modifications en ALGOL 68. Il existe d'autres méthodes présentant des caractéristiques différentes [WILLIS]. Une méthode récursive par exemple, permettrait une programmation plus concise, mais moins efficace.

Nota : Les modes traités par cet algorithme sont supposés remplir toutes les conditions concernant les modes qui ont, auparavant, été vérifiées pour chacun d'entre eux. De plus l'équivalence de mode a été réalisée ainsi que la réduction des unions imbriquées et la mise en ordre des sous-modes de chaque mode union.

Par "réduction des unions imbriquées" il faut comprendre : si un sous-mode d'une union est lui-même une union, cette union interne va disparaître et ses sous-modes vont être des sous-modes à part entière de l'union englobante.

Ex : *union (réel, union (ent, bool), rep compl)* se transforme en
union (réel, ent, bool, rep compl).

III.5 - ANALYSE SYNTAXIQUE -

1 - LE PROBLEME -

Dans le rapport de définition du langage Algol 68, les deux aspects de la syntaxe, syntaxe hors-contexte et syntaxe contextuelle, sont exprimés à l'aide d'un formalisme unique, les W-grammaires, ce qui présente l'avantage d'une plus grande homogénéité. Il n'existe malheureusement pas d'analyseurs généraux de W-grammaires. L'écrivain de compilateurs est donc conduit à utiliser les méthodes classiques d'analyse, basées sur une grammaire hors-contexte du langage. Selon la méthode d'analyse utilisée, la grammaire doit avoir certaines propriétés plus ou moins restrictives. Les aspects contextuels de la syntaxe sont traités par le biais d'"actions de compilation". Ce sont des procédures dont les points d'appel sont insérés dans la grammaire, et qui sont appelées au cours de l'analyse syntaxique. Ces méthodes sont désormais classiques [MUNICH], et supposent l'existence d'une grammaire hors-contexte du langage. Examinons les problèmes posés par l'obtention d'une grammaire hors-contexte d'Algol 68 en vue de l'analyse.

Il est possible d'obtenir automatiquement une grammaire hors-contexte à partir de la W-grammaire du Rapport [BUFFET]. Cette méthode présente l'avantage de garantir que la grammaire obtenue définit un sur-ensemble du langage défini par la W-grammaire. Par contre, sa forme a peu de chances d'être adaptée à l'utilisation dans un compilateur. Pour satisfaire aux exigences d'adaptation à une méthode particulière, il est plus simple de procéder à la main. La transcription depuis la W-grammaire est assez systématique. Elle consiste essentiellement en l'abandon des métanotions qui traduisent les aspects contextuels de la syntaxe (MODE, NEST), et en un réarrangement local des règles obtenues. Les risques d'erreur ne sont pas supérieurs à ceux provenant d'une transformation manuelle de la grammaire produite automatiquement, ce qui aurait sans doute été nécessaire. De plus, ces erreurs sont assez aisément détectables. Le vrai problème réside dans la nature de la grammaire obtenue.

La grammaire hors-contexte définit un langage qui n'est pas le langage Algol 68. Sauf erreur, on a la grammaire d'un sur-ensemble du langage Algol 68. Dans le cas de langages définis par une grammaire hors-contexte, comme Algol 60, on dispose d'une grammaire hors-contexte qui définit le langage concerné. Le niveau de définition utilisé pour Algol 68 conduit à une définition plus précise

du langage, mais malheureusement plus éloignée d'une implantation. Deux problèmes se posent à l'implanteur lors de l'écriture d'une grammaire hors-contexte d'Algol 68.

1 - Elle ne peut pas rendre compte de la priorité des opérateurs, car celle-ci dépend d'un mécanisme d'identification, qui n'est pas exprimable par une grammaire hors-contexte. Il n'apparaît dans la grammaire qu'un seul terminal, *indicateur de priorité*, au lieu des dix correspondant chacun à une priorité, selon le principe utilisé dans la grammaire d'Algol 60 ou d'Algol W. Il s'ensuit que l'arbre d'analyse ne traduit pas la priorité des formules et s'éloigne donc de manière importante de la structure d'arbre abstrait correspondante, ce qui est gênant pour la génération de code.

Une solution à ce problème consiste à imposer l'écriture des déclarations de priorité avant l'utilisation des opérateurs correspondants dans la région. L'analyseur lexicographique dispose des tables de priorité par région et peut retourner 10 symboles terminaux différents pour chacune des priorités d'opérateur. La grammaire peut alors exprimer la priorité des formules. C'est la solution généralement adoptée pour les compilateurs en un passage. Nous faisons un passage préalable qui collecte ces informations.

2 - Elle contient des ambiguïtés qui ne s'éliminent qu'au prix d'une perte de structure de la grammaire. En effet, des constructions sémantiquement distinctes peuvent avoir une forme externe commune, d'où l'ambiguïté de la grammaire.

Exemples :-saut et appel de procédure sans paramètre ... *;* l *;* ... peut être un appel de la procédure l ou signifier *aller a* l, puisque le symbole *aller a* peut être omis.

-appel de procédure à paramètres et indexation ... $p(i)$ ... peut être un appel de la procédure p ou une indexation du tableau p.

-déclaration d'identité (de variable) et formule.
; $m\ x = \ldots$ *;* peut être une déclaration d'identité de la variable x utilisant l'indicateur de mode m ou une formule utilisant l'opérateur unaire m.

Le même problème se pose avec la déclaration de variable *;* $m\ x$ *;* ou *;* $m\ x := \ldots$ *;*

Dans tous les cas, l'ambiguïté est liée à un processus d'identification. Il suffit pour la lever, d'imposer dans le programme la présence de toute déclaration avant son utilisation, et, dans le cas des sauts, de rendre le symbole *aller a* obligatoire. Dans cette optique, la déclaration des modes et procédures

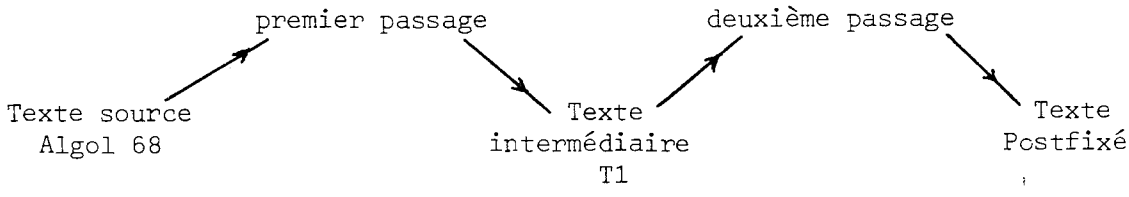
récursifs devient délicate - il faut bien déclarer l'un avant l'autre - et on peut être amené à introduire dans le langage des déclaraticns préliminaires [CUR-68R], ou des pragmats [BANATRE].

Une autre solution consiste à n'effectuer l'analyse syntaxique que lorsque tous les objets sont identifiés, ce qui suppose une certaine reconnaissance préliminaire du texte. Signalons à ce sujet deux approches intéressantes, celle de l'équipe de Manchester, qui module le nombre de passages selon le type du programme - les "bons" programmes ne sont pas pénalisés -, et celle de l'équipe Rennaise qui met en attente les processus qui dépendent d'une information manquante.

2 - NOTRE APPROCHE DE L'ANALYSE SYNTAXIQUE -

Etant donné que le langage Algol 68 sans restriction s'accomode mal des méthodes d'analyse classiques, nous avons décidé de faire précéder l'analyse syntaxique par un passage de reconnaissance dont le rôle est de collecter le maximum d'information pour permettre ensuite une analyse commode . La méthode d'analyse choisie est la méthode LL(1), qui est une méthode descendante et sans retour-arrière [FOSTER]. C'est une méthode simple et efficace, permettant l'insertion commode d'actions de compilation. Nous disposons à Grenoble d'un système de production d'analyseurs basé sur ce principe [GRIFF-2], qui a été modifié pour s'adapter à nos besoins [BRE-DEL].

Nous avons donc un schéma d'analyse en deux passages. Le premier réalise la discrimination des symboles qui posent des problèmes à l'analyse LL(1). Il produit un texte T1 qui est une image du texte source où les symboles "délicats" sont différenciés. Le deuxième passage réalise l'analyse syntaxique effective de ce texte T1.



- Les contraintes de l'analyse LL(1) -

Le principe de la méthode est le suivant : partant de l'axiome de la grammaire, et explorant le texte à analyser de gauche à droite en lisant à chaque pas seulement un symbole (le symbole courant), on décide pour chaque règle quelle alternance appliquer, en fonction seulement du symbole courant. Lorsque ce parcours de la grammaire passe sur un symbole terminal, qui doit correspondre au symbole courant, un nouveau symbole du texte source est lu et il devient symbole courant. Un choix unique d'alternance dans une règle ne peut être fait en fonction du seul symbole courant, que si les ensembles de symboles terminaux commençant les alternatives d'une même règle, forment des ensembles disjoints. Knuth énonce quatre conditions auxquelles doit satisfaire une grammaire pour être LL(1) [KNUTH-3] ; elles ont été condensées en une seule par M. Griffiths [GRIFF-2] : les "symboles directeurs" des alternances d'une règle doivent constituer des ensembles disjoints.

Une des conséquences en est que la grammaire ne doit pas contenir de récursivité à gauche. Celles-ci doivent être transformées - automatiquement ou à la main - en récursivités à droite. Les méthodes de mise au point de grammaires LL(1) ont été étudiées par J. Bordier [BORDIER]. Nous présentons maintenant les principales sources de problèmes pour une analyse LL(1) d'Algol 68, qui sont les indicateurs, les déclareurs et les parenthèses [68TECH].

- Les indicateurs -

Les indicateurs de mode et de priorité peuvent conduire à des constructions ambiguës. Cette ambiguïté n'est pas intrinsèque, c'est-à-dire qu'un regroupement adéquat des règles contenant ces constructions semblables, permet de l'éliminer. Les constructions du type *indicateur identificateur* seraient donc considérées par une telle grammaire soit comme des déclarations uniquement, ou comme des formules uniquement. Un tel regroupement a été fait lors de l'étude d'une grammaire de précédences d'Algol 68 [REZIG]. Mais que dire d'une grammaire utilisée dans un compilateur, où sont regroupées des constructions aussi dissemblables que déclaration et formule !

Nous avons choisi de faire une reconnaissance préalable des indicateurs au cours du premier passage, de manière à disposer lors du second passage de deux symboles distincts :

- indicateur de mode
- indicateur de priorité.

- Les parenthèses -

La multiplicité d'emploi des parenthèses rend quasi impossible l'analyse LL(1) d'un texte Algol 68, à moins d'imposer des restrictions à leur usage - la première de ces restrictions étant l'emploi des crochets [ ] pour les tableaux, crochets malheureusement inexistants sur de nombreux matériels.

Le premier passage assure la reconnaissance de certaines parenthèses :

- parenthèse de proposition fermée ou collatérale,
- parenthèse de proposition conditionnelle ou cas,
- parenthèse de déclareur de tableau,
- parenthèse de vacuum,
- parenthèse d'élément de cas union.

Les autres types de parenthèses n'ont pas besoin d'être reconnus car le contexte gauche permet de les différencier { *struct* (..., *proc* (..., *p*(... }.

- Les déclareurs -

Les déclareurs sont employés dans différentes constructions syntaxiques. Ces constructions sont souvent différenciées par le contexte droit du déclareur. Or en analyse LL(1), on dispose seulement du contexte gauche d'une construction.

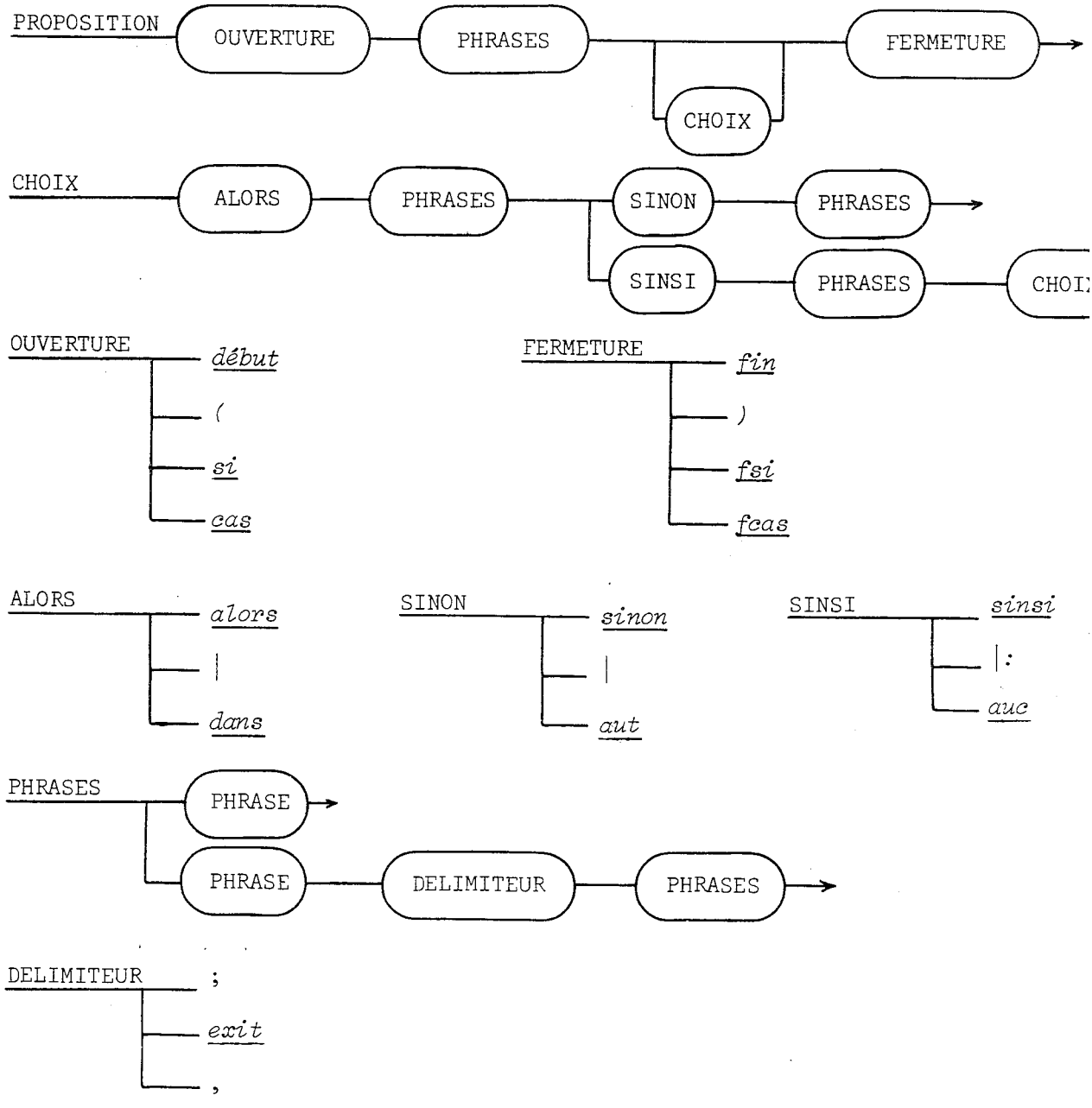
Dans le texte T1 produit par le premier passage, les déclareurs sont précédés par un préfixe, caractéristique du déclareur. On différencie les constructions suivantes :

- déclaration d'identité,
- déclaration de variable,
- forceur,
- générateur,
- texte de routine.

La discrimination de ces constructions ou symboles est réalisée par le premier passage du compilateur, dont nous présentons maintenant les caractéristiques.

3 - LE PREMIER PASSAGE -

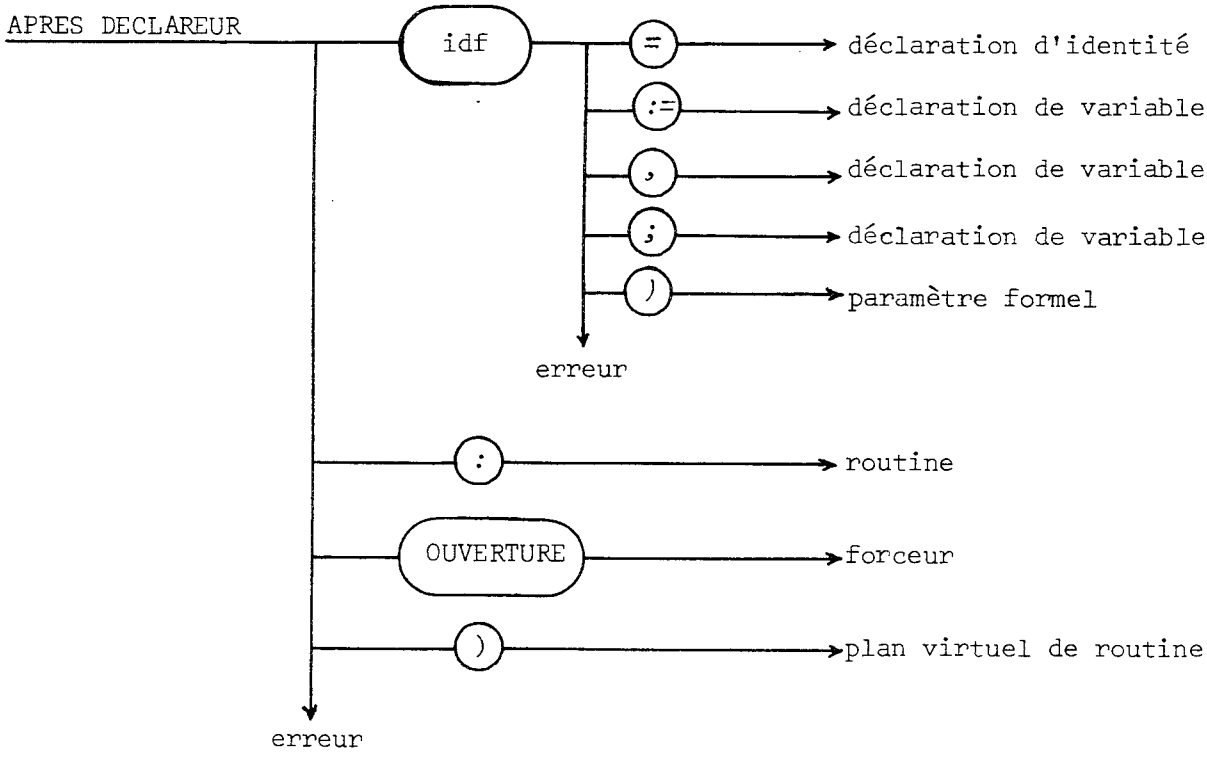
Le premier passage reconnaît la structure du programme à analyser, sans entrer dans le détail de la plupart des constructions. Pour illustrer cette notion, nous donnons le diagramme qui correspond à l'analyse des propositions (proposition fermée, proposition conditionnelle, proposition cas entier et proposition cas union).



La construction PHRASE correspond à tout ce qui peut être déclaration ou unité, sans les différencier à ce niveau. Un tel diagramme décrit donc un sur-ensemble assez large des programmes Algol 68. Il est suffisant pour reconnaître la structure de région, ce qui permet la construction des tables d'indicateurs par région.

Les autres fonctions du premier passage sont exprimées à l'aide du même type de diagrammes. A titre d'exemple, nous donnons celui qui correspond à la discrimination des déclareurs.

Un déclareur est reconnu par son contexte droit, ce qui est exprimé par le diagramme ci-dessous :



Le premier passage est écrit en CDL (Compiler Description Language), [KOSTER-2] un langage d'écriture de compilateurs, qui se prête bien à l'expression de tels diagrammes.

Un analyseur lexicographique reconnaît les symboles terminaux comme "identificateur", les mots clés et les opérateurs. Il a pour particularité d'être multilingue, c'est-à-dire d'accepter les mots clés d'Algol 68 exprimés dans une langue quelconque, moyennant bien entendu l'initialisation adéquate. Il est de plus facilement adaptable pour d'autres langages de programmation [68TECH].

Le texte produit par le premier passage est ensuite fourni en entrée au second passage. Il se pose le problème des erreurs détectées au cours de ce passage. Nous n'avons pas trouvé de solution générale satisfaisante. Un message est émis, mais le texte produit n'est corrigé que lorsque la correction apparaît sûre. Dans les autres cas le texte produit contient l'erreur, et l'analyseur du deuxième passage met en action un mécanisme plus général de récupération d'erreurs.

4 - LE DEUXIEME PASSAGE -

Le deuxième passage a pour rôle de réaliser l'analyse syntaxique du programme source, ou plus exactement de son image produite par le premier passage. La discrimination effective des indicateurs, déclareurs et parenthèses, est faite au niveau de l'analyseur lexicographique, qui délivre des codes différenciés. Nous montrons comment se fait cette discrimination, puis nous étudions les problèmes posés par la construction de l'arbre abstrait.

4.1 - Analyseur lexicographique du deuxième passage -

L'analyseur lexicographique du second passage "lit" symbole par symbole le texte T1 produit par le premier passage. En fonction des informations recueillies au cours du premier passage sur la nature de certains symboles, il peut délivrer des codes distincts pour des symboles initialement identiques. Les symboles concernés sont :

- déclareurs -

Les déclareurs autres que les déclareurs de tableau sont précédés dans T1 par un préfixe de déclareur qui est suivi d'un déplacement dans une table des déclareurs. C'est dans cette table qu'est mis le type du déclareur lorsqu'il est reconnu au premier passage. Selon la nature du déclareur, l'analyseur lexicographique retourne le code :

- préfixe de déclaration d'identité,
- préfixe de déclaration de variable locale,
- préfixe de déclaration de variable globale,
- préfixe de générateur local,
- préfixe de générateur global,
- préfixe de déclareur de routine,
- préfixe de forceur.

- indicateurs -

Dans le texte T1, un indicateur est suivi du numéro d'indicateur qui permet de le retrouver dans les tables de déclarations par région, construites au premier passage, et d'un déplacement dans la table des déclareurs, utile au cas où il s'agit d'un indicateur de mode.

Les codes délivrés sont, après identification :

- indicateur de priorité,
- indicateur de déclaration d'identité,
- indicateur de déclaration de variable locale,
- indicateur de déclaration de variable globale,
- indicateur de générateur local,
- indicateur de générateur global,
- indicateur de routine,
- indicateur de forceur.

- parenthèses -

Dans le texte T1, les parenthèses sont accompagnées de leur numéro de bloc potentiel qui indexe une table des parenthèses où se trouvent, entre autres, des informations concernant leur nature :

- le type de la parenthèse,
- le numéro de déclareur, pour le cas où la parenthèse peut commencer un déclareur de tableau,
- le numéro d'indicateur suivant la parenthèse fermante, pour le cas où le type de la parenthèse est indéterminé. La nature de l'indicateur permet alors de lever l'indétermination.

Les codes délivrés sont :

- parenthèse de vacuum,
- parenthèse de plan formel de routine,
- parenthèse de proposition fermée ou collatérale,
- parenthèse de proposition choix,

- parenthèse d'élément de cas union, et, dans le cas où la parenthèse commence un déclareur de tableau :
- (- déclaration d'identité
- (- déclaration de variable locale
- (- déclaration de variable globale
- (- générateur local
- (- générateur global
- (- routine
- (- forceur

4.2 - Production de l'arbre abstrait et analyse syntaxique -

L'analyse syntaxique du texte T1 produit par le premier passage est faite selon un schéma LL(1). Le texte T1 est aisément analysable par cette méthode car les symboles "délicats" sont maintenant différenciés. Un des intérêts de l'analyseur syntaxique est, bien entendu, de vérifier la correction des programmes relativement à une grammaire hors-contexte ; il sert également de support à des actions de compilation (que nous avons soulignées dans les règles ci-après).

Une des fonctions de ces actions de compilation est la production d'un texte intermédiaire, qui est l'arbre abstrait sous forme postfixée. Le sous-arbre associé à une construction, est décrit comme attribut synthétisé par cette construction. La transcription de ce mécanisme sous forme d'actions de compilation insérées dans une grammaire, se fait de la manière suivante :

- les actions associées à une alternance d'une règle construisent le sous-arbre correspondant à cette alternance,
- les opérandes d'un noeud sont produits au fur et à mesure de leur rencontre, et le noeud est créé ensuite.

En fait, le principe est élémentaire, et se traduit a priori par l'insertion d'une action de création de noeud pour chaque construction.
Exemples : considérons les règles syntaxiques de l'affectation et de la proposition conditionnelle.

a) Affectation

Les actions de la règle "tertiaire" auront créé le sous-arbre correspondant, de même pour "unité". L'action "créer noeud (:=)" met donc en place le noeud postfixé (:=) après les opérandes : T U (:=)
où T et U représentant les sous-arbres associés à tertiaire et unité respectivement.

b) proposition conditionnelle ← si, proposition sérielle$_1$, alors,
proposition sérielle$_2$, choix, fsi, créer_noeud_(choix-booléen)
choix → ∅, créer_noeud_(skip)
→ sinon, proposition sérielle$_3$

Si PS_i représente le sous-arbre créé par la règle proposition sérielle$_i$, l'arbre créé pour proposition conditionnelle a l'une des deux formes suivantes :

PS_1 PS_2 PS_3 (choix-booléen)

ou

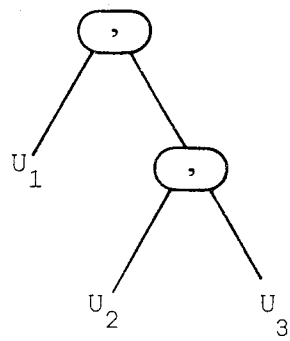
PS_1 PS_2 (fant) (choix-booléen)

La forme des règles et l'emplacement des actions de compilation régissent la forme de l'arbre. Ainsi, une liste d'unités U_1, U_2, U_3 analysée par la règle :

(1) - U-liste → U
→ U,',',U-liste, créer_noeud_(,)

donne l'arbre

U_1 U_2 U_3 (,) (,) soit

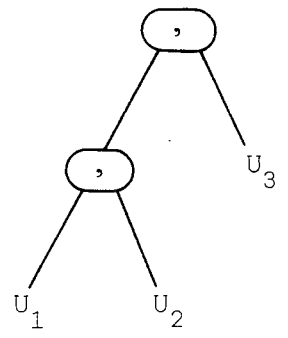


Analysée par la règle :

(2) - U-liste → U
→ U-liste , ',' , U, créer_noeud_(,)

il lui correspond l'arbre :

U_1 U_2 (,) U_3 (,) soit



Ceci étant, il est toujours possible de transformer une récursivité à gauche en récursivité à droite, en conservant le sens des actions de compilation.

Une telle transformation peut être réalisée automatiquement [BORDIER]. Cette transformation présente un intérêt certain dans le cas des formules. En effet, la règle d'évaluation des formules en Algol 68, s'exprime naturellement par une récursivité à gauche :

formule → formule, opérateur, opérande.

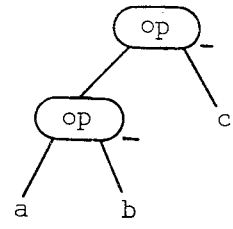
Ce qui signifie que l'expression $a - b - c$ correspond à $(a - b) - c$ et non $a - (b - c)$.

Or la règle :

(1) formule → formule, opérateur, opérande, créer noeud (op)
→ opérande

produit l'arbre

a b - c - soit

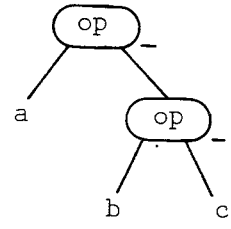


tandis que la règle

(2) formule → opérande, opérateur, formule, créer noeud (op)
→ opérande

produit l'arbre

a b c - - soit



(1) contient une récursivité à gauche, qui doit être éliminée dans une grammaire LL(1).

(2) est LL(1) moyennant la factorisation de *opérande*, mais produit le mauvais arbre.

On peut trouver une formulation des règles telle que l'on puisse obtenir l'un ou l'autre arbre en changeant simplement la place des actions de compilation dans la règle.

Exemples :

U-liste → U, fin de liste

(1) fin de liste → ∅

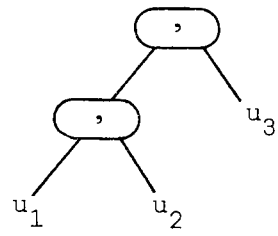
→ ',' , U, créer noeud(,), fin de liste

(2) fin de liste → ∅

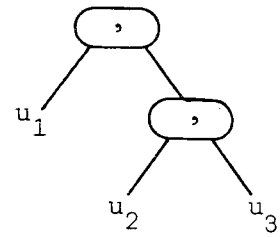
→ ',', U, fin de liste, créer noeud (,)

Pour la phrase u_1, u_2, u_3
Les règles (1) et (2) construisent les arbres respectifs :

(1) u_1 u_2 , u_3 , soit



(2) u_1 u_2 u_3 , , soit



Le même schéma peut s'appliquer aux formules :

formule → opérande, fin de formule

(1) fin de formule → ∅

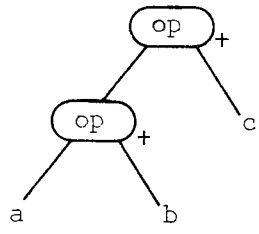
→ opérateur, opérande, créer noeud (op),
fin de formule

(2) fin de formule → ∅

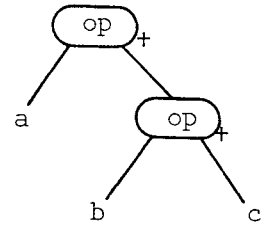
→ opérateur, opérande, fin de formule,
créer noeud (op)

pour la formule a + b + c
les règles (1) et (2) construisent les arbres respectifs :

(1) a b + c + soit



(2) a b c + + soit



4.3 - Priorité des opérateurs -

Les opérateurs ont chacun une priorité que l'utilisateur choisit par déclaration (sa valeur est de 1 à 9) et qui est imposée pour les opérateurs unaires (sa valeur est 10). Cette priorité est attachée à l'opérateur au moment de l'analyse syntaxique. Nous pouvons donc dans la grammaire considérer 10 symboles terminaux distincts op1, op2, ... op10, pour chacune des classes de priorité.

Le moyen le plus naturel d'exprimer la syntaxe des formules, consiste à utiliser une grammaire avec récursivité à gauche. Si opérateur$_i$ représente un opérateur de priorité i, on a la grammaire suivante :

tertiaire ← formule 1

formule 1 ← formule 1, opérateur 1, formule 2, créer_noeud (op)
← formule 2

et pour i variant de 2 à 9, une règle de la forme :

formule i ← formule i, opérateur i, formule i+1, créer_noeud (op)
← formule i+1

formule 10 ← opérateur 10, formule 10, créer_noeud (op)
← secondaire

Dans ce cas, la forme de l'arbre syntaxique est la même que celle de l'arbre abstrait. Mais une grammaire avec récursivité à gauche n'est pas LL(1) et on doit écrire la grammaire en utilisant la récursivité à droite.

tertiaire → formule 1

formule 1 → formule 2, fin de formule 1

fin de formule 1
→ ∅
→ opérateur 1, formule 2, créer_noeud (op), fin de formule 1

et pour tout i de 2 à 9, une règle de la forme :

formule i → formule i+1, fin de formule i

fin de formule i

→ ∅

→ opérateur i, formule i+1, créer_noeud (op), fin de formule i

formule 10 → opérateur 10, formule 10, créer_noeud (op)

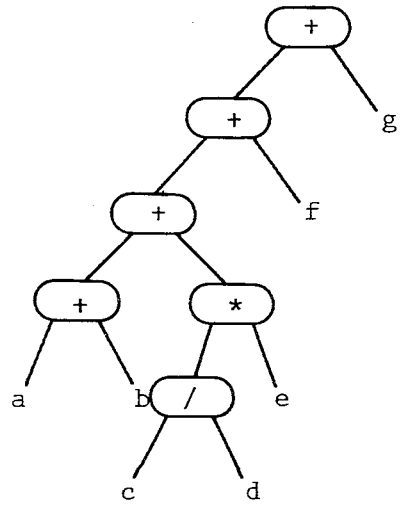
→ secondaire

Exemple : *a + b + c / d * e + f + g*

l'arbre produit a les représentations suivantes :

a b + c d / e * + f + g +

ou



Dans notre grammaire, pour éviter une telle multiplication de règles, nous avons écrit la syntaxe des formules de la manière suivante :

tertiaire → début_de_formule, formule, fin_de_formule

formule → opérande, traiter_opérateur_binaire, opérateur, formule

opérande → traiter_opérateur_unaire, opérateur, opérande ;

→ secondaire

Les actions de compilation début_de_formule, fin_de_formule, traiter_opérateur_binaire, et traiter_opérateur_unaire prennent en compte la priorité des opérateurs et produisent l'arbre correct à l'aide d'une pile annexe.

4.4 - La grammaire -

Le principe est simple. La pratique est rendue un peu plus complexe du fait de la forme de la grammaire,et parce que l'arbre est partiellement décoré. En effet, à chaque noeud de l'arbre est associé un numéro d'unité qui donne au programmeur un repère dans le texte source (localisation des messages, aide à la mise au point). Ce numéro d'unité est représenté au cours de l'analyse

par une variable globale qui est mise à jour à la rencontre de certains symboles délimiteurs. Or, au moment où est effectivemat créé le noeud, la variable globale qui représente le numéro d'unité ne correspond pas au numéro d'unité du noeud. Une solution consiste à empiler ce numéro au moment adéquat. Lors de la création du noeud, cette information est dépilée.

D'une manière générale, à chaque construction est associée une action qui empile le noeud lui-même,et des informations annexes. Au moment de la création du noeud, une fonction de création, créer noeud, prend les éléments au sommet de la pile et crée le noeud avec le nombre adéquat d'informations annexes.

Nous donnons à titre d'exemple la règle correspondant à l'affectation

affectation → tertiaire, traiter affectation, :=, unité, créer noeud1

```
proc traiter affectation = neutre : co empiler le noeud (:=) et
                                       son numéro d'unité
                                   co

proc créer noeud1        = neutre : co créer noeud sommet de pile avec
                                       1 paramètre
                                   co
```

Le texte postfixé produit a la forme suivante :

texte du tertiaire, texte de l'unité, noeud (:=) et numéro d'unité.

Nous ne précisons pas ici la grammaire utilisée. Pour qu'elle soit utile, il faudrait y joindre l'explication de toutes les actions de compilation, ce qui dépasserait le cadre de cette thèse. Nous le faisons dans les notices techniques du compilateur.

5 - CONCLUSION -

L'analyse syntaxique d'Algol 68 sans restrictions n'est pas un problème simple. Nous pensons que la forme externe du langage a une certaine importance, même si elle n'en constitue pas l'essentiel. Dans le cas d'Algol 68, nous aurions souhaité que des constructions sémantiquement distinctes aient une forme externe différente. L'emploi systématique des crochets pour les tableaux faciliterait certes la lecture des programmes et leur analyse. Le préfixage des déclarations de variables, aurait l'avantage de rappeler que *ent i* n'a pas la même signification en tant que paramètre de procédure et en tant que déclaration de variable, et aussi d'éviter la confusion avec les formules unaires dans le cas de l'emploi d'indicateurs. L'emploi d'un

symbole distinctif dans la déclaration d'identité éviterait cette même confusion (*réel pi ≡ 3.14*).

De tels aménagements de détail permettraient en outre une détection d'erreurs plus efficace en améliorant la qualité des messages émis, et en facilitant la continuation de cette détection après une erreur. Il est significatif que de nombreux écrivains de compilateurs aient dû imposer quelques restrictions au langage, pour pouvoir utiliser une méthode d'analyse satisfaisante.

III.6 - LA GENERATION DE CODE -

La description de la fonction de traduction donnée précédemment a permis de dégager les fonctions de la traduction et celles d'un compilateur pour le langage Algol 68. Elle ne suppose aucune solution technique particulière et fait abstraction d'éventuelles particularités au niveau du matériel pour la génération de code. A partir de cette description, si nous voulons donner des spécifications complètes pour le compilateur, il nous faut définir des transformations permettant d'atteindre le jeu d'instructions et de données du calculateur réel. Nous pouvons remarquer que la conception est toujours descendante alors qu'habituellement, à ce niveau,elle devient ascendante.

Les aspects abordés ici, concernent essentiellement :

a) - la représentation et la gestion des valeurs en Algol 68 pour une classe de calculateurs. Le choix des solutions techniques tient compte au maximum des possibilités de portabilité,et limite à quelques processus, situés au bas de la hiérarchie, et en fin de traduction, la prise en compte des particularités technologiques du matériel.

b) - la mise en oeuvre d'un ensemble d'optimisations,surtout locales,sur les sous-arbres de l'arbre abstrait décoré dont le parcours est décrit par une action collatéral ... fincollatéral.

c) - la production de code, à partir des ensembles de quadruplets, par transformations successives.

Nous rappelons tout d'abord les principaux aspects nouveaux apportés par le calculateur dans la représentation de la fonction de traduction. On trouve une illustration des points envisagés et des compléments à cette étude dans les notices techniques du compilateur implanté dans le calculateur IBM 360 [68TECH].

1 - LE CALCULATEUR ET LA GENERATION DE CODE -

L'introduction des caractéristiques d'une classe de matériels dans lesquels le code objet s'exécute,a deux conséquences pour le code engendré. D'une part, des optimisations vont être possibles dans la gestion mémoire et dans la manipulation de certaines valeurs. D'autre part,pour obtenir une implantation efficace, on adapte la représentation des valeurs de mode standard aux parti-

cularités technologiques du calculateur. (Rappelons que les valeurs de mode standard sont utilisées dans la construction de toutes les autres valeurs).

L'insuffisance de la technologie nécessite, en général, au niveau de la production de code, l'introduction d'un traducteur produisant pour chaque quadruplet introduit, une séquence d'instructions (exécutables pour cette machine). Par cette méthode, nous rendons le compilateur partiellement portable, seuls l'initialisation des tables pour les modes standard, et le traducteur, étant à modifier si l'on veut changer de calculateur.

Nous établissons une classification des différents aspects technologiques intéressant la génération de code. Nous avons séparé ceux que l'on peut retrouver sur une large classe de calculateurs, de ceux qui restent plus particulièrement attachés à une machine. Notre étude n'est pas exhaustive et ne donne pas un panorama complet des possibilités offertes à l'heure actuelle ; toutefois, pour un matériel donné, elle peut constituer un guide intéressant pour définir la forme du code à produire.

Les trois aspects envisagés pour le calculateur sont :

- les possibilités de calculs simultanés (parallélisme technologique),
- les ressources mémoire,
- le jeu d'instructions et les possibilités de micro-programmation.

Des techniques plus spécifiques, telles que la gestion automatique de pile, bien qu'elles modifient assez profondément la structure du code produit sont à rapprocher du troisième aspect envisagé.

1.1 - Calculs simultanés -

Lorsque le calculateur dispose de plusieurs unités de calcul il peut être intéressant d'envisager la production de séquences de code pouvant s'exécuter simultanément sur ces unités. Le parcours collatéral de génération permet de déterminer les parties de programme concernées, et les points auxquels les séquences de contrôle de l'exécution simultanée doivent être insérées. Toutefois il est nécessaire que (pour l'ensemble des unités concernées) les rapports moyens de taille et de temps d'exécution entre les séquences de contrôle d'une part, et les séquences utiles à l'évaluation sémantique du programme d'autre part, restent raisonnables, pour que l'utilisation de cette simulta-

néité ait un intérêt. Ce choix n'est peut être pas simple à réaliser et il peut nécessiter des modifications de l'arbre abstrait. En conséquence, cette possibilité de calcul simultané peut être considérée comme une optimisation globale à effectuer sur l'arbre abstrait du programme.

1.2 - Les ressources mémoires -

Dans un calculateur on établit habituellement une hiérarchie dans les ressources mémoire : mémoires périphériques, mémoire centrale, et une mémoire que nous qualifions de rapide. Nous ne parlons pas de technologies plus sophistiquées utilisant par exemple, une antemémoire.

Pour un compilateur on retrouve la même hiérarchie, même si une adéquation entre les ressources physiques et les ressources utilisées par le système à l'exécution, est nécessaire.

Dans un compilateur classique, au niveau de la représentation de l'environnement d'évaluation :

- les mémoires périphériques ne sont concernées que par les entrées-sorties
- la mémoire que nous qualifions de centrale, même s'il s'agit d'une mémoire virtuelle, est utilisée par la pile et le tas,
- la mémoire rapide, que l'on retrouve sur des machines type IBM 360 est formée d'un ensemble de ressources privilégiées souvent appelées registres, et est utilisée dans la représentation de la zone d'évaluation.

Ce choix est justifié dans la mesure où les informations contenues dans la zone d'évaluation ne sont utilisées que localement, et où leur accès doit rester très rapide.

Les registres ne servent donc pas uniquement, sur la classe de machines que nous considérons, de ressources de calcul ou d'accumulateurs mais également de sauvegarde temporaire pour les informations locales. Si le calculateur dispose d'une mémoire rapide, à accès privilégié par rapport à la mémoire centrale, elle peut également être utilisée pour la représentation de la zone d'évaluation.

Remarques :

1 - Sur le calculateur IBM 360, les registres servent aussi de base pour les accès aux objets manipulés.

2 - En Algol 68 les entrées-sorties n'utilisent pas comme support que les mémoires périphériques, mais également la mémoire centrale.

1.3 - Les instructions du calculateur -

Les particularités de l'ensemble des instructions d'un calculateur ont de nombreuses implications.

a) - Les valeurs de mode standard -

On s'efforce d'adapter leur représentation aux instructions du calculateur, de telle sorte qu'une seule instruction permette de manipuler, ou d'opérer, sur l'ensemble de cette représentation. On peut alors disposer du maximum de précision pour ces valeurs, tout en bénificiant d'un code optimal (tant en place pour le codage de l'opération, qu'en temps d'exécution). Ainsi, en fonction du jeu d'instructions du calculateur, on est amené à fixer un nombre maximum pour les différentes longueurs des valeurs définies dans le prélude standard d'Algol 68. A partir de ce nombre, toutes les valeurs du programme de longueur supérieure ont une même représentation, et une même précision sur ce calculateur (ce qui ne préjuge pas de la représentation sur un autre calculateur). Ce choix est enfin harmonisé en fonction des modifications d'élargissement possibles entre ces diverses valeurs de mode standard. Signalons qu'il est possible de définir des préludes particuliers pour tenir compte au maximum du jeu d'instructions, aux dépens toutefois de la portabilité des programmes les utilisant.

Exemple : Considérons le cas du calculateur IBM 360.
Les valeurs de mode entier court, entier, entier long,
réel court, réel, réel long,
sont représentées dans notre compilateur par :

entier court : demi-mot, réel court : mot,
entier : mot, réel : double-mot,
entier long : mot, réel long : double-mot,

toutes les autres "longueurs" ont la même représentation que l'entier long ou le réel long.

b) - Représentations optimales des valeurs sur un calculateur donné -

L'utilisation de toutes les possibilités offertes par le jeu d'instructions d'un calculateur, nous amène à redéfinir certaines représentations de valeurs pour rendre leur traitement plus efficace, et à distinguer certains noeuds de l'arbre abstrait afin d'engendrer une meilleure séquence de code. Ces optimisations ne sont dues qu'au calculateur, et portent, tant sur la rapidité d'exécution des instructions produites, que sur la réduction de l'espace nécessaire à leur codage ou à la représentation des valeurs.

Nous allons examiner quelques cas classiques : les valeurs de type accès, les valeurs booléennes et certaines constantes. Nous envisageons aussi l'utilisation d'instructions spécifiques de certains calculateurs.

- Les valeurs de type "accès" -

Dans la plupart des jeux d'instructions, il existe une possibilité pour manipuler des accès, en général sous forme de chargement dynamique d'adresse dans un registre. On peut, par suite, définir une représentation plus adaptée pour les valeurs de mode repère. Elle nécessite moins d'espace mémoire, et les traitements portant sur cette valeur sont rendus plus rapides par l'utilisation d'une telle instruction.

Exemple : Sur le calculateur IBM 360 l'instruction : LA R, A
où A est l'adresse d'un objet en mémoire, permet de charger dans le registre R l'adresse de cet objet.
Si l'on considère le programme Algol 68 :

```
début
    ent ri, rep ent rri ;
    ent i = 1 ;...
    rri := ri
fin
```

Les objets *ri* et *i*, bien que de mode *rep ent* et *ent*, ont la même représentation. Lorsque l'on veut élaborer l'affectation *rri := ri* l'instruction LA permet de charger dans le registre R la valeur de mode *rep ent*.

Nous étudions dans la suite (voir paragraphe 2), les conséquences d'une telle possibilité qui représente une optimisation importante pour la représentation des valeurs de mode repère.

- Les valeurs booléennes -

Certaines instructions positionnent des indicateurs de condition (par exemple les comparaisons). Si la valeur booléenne associée est utilisée pour la suite de l'évaluation, elle se trouve représentée sous une forme particulière qu'il peut être intéressant de conserver. Par exemple, si elle contrôle un branchement, on peut éviter deux transformations dynamiques consécutives de sa représentation : représentation par un indicateur de condition en représentation en mémoire, et vice-versa, pour effectuer le branchement.

Par exemple considérons la séquence Algol 68:

```
début
    ent a, b, c ;
    ...
    c := si a > b alors 1 fsi ;
    ...
fin
```

La comparaison entre a et b s'effectue par une instruction C R, A où R est un registre, A l'adressage d'un objet en mémoire. Elle positionne le code condition. L'élaboration du choix booléen conduit à exécuter, suivant la valeur de ce code condition, une instruction de branchement.

- Les constantes -

Les calculateurs disposent en général d'instructions dites à opérandes immédiats. Ces instructions permettent d'éviter un accès dynamique à une table des constantes, en insérant dans le codage même de l'instruction la représentation de la valeur opérande. Le format de l'instruction impose toutefois certaines restrictions quant au mode de la valeur.

Exemple : Dans le calculateur IBM 360 les octets peuvent être manipulés directement par des instructions à opérandes immédiats. On peut ainsi éviter de ranger les constantes booléennes (représentées par un octet) en table, en les plaçant dans le code de l'instruction qui les utilise.

- Mise en oeuvre d'instructions particulières -

Il existe des instructions spécifiques de certains calculateurs,qui permettent des optimisations locales lorsque l'on peut détecter pendant le parcours de génération,les sous-arbres adéquats. Signalons à titre d'exemple le "branchement conditionnel avec incrémentation" qui permet de "contrôler" en une seule instruction les itérations à pas positif (cas que l'on peut rencontrer relativement fréquemment). Ces optimisations compliquent toutefois la génération de code,mais ne perturbent pas la structure de son architecture. Tout se passe comme si l'on introduisait des noeuds supplémentaires dans l'arbre abstrait, en vue d'une interprétation dans une machine donnée. Ceci montre les limites de ces "optimisations" qui, si elles étaient systématisées, feraient perdre tout intérêt à notre effort de structuration, et donneraient une autre approche plus pragmatique et bien peu systématique de la compilation, sans souci de portabilité.

En conclusion, malgré les quelques points que l'on vient d'étudier, et qui sont illustrés au niveau des notices techniques du compilateur, le jeu d'instructions sert surtout à représenter les quadruplets sous une forme exécutable. Il semble dans ce domaine que la microprogrammation pourrait offrir des perspectives intéressantes,en adaptant, comme le propose Lindsey [LINDSEY], la technologie au langage. Les quadruplets pourraient alors être interprétés de manière plus efficace (gain de place pour le codage et gain de temps d'exécution). La traduction "quadruplets - séquence d'instructions" pourrait,moyennant ces aménagements,se transformer en une traduction "quadruplets - ensemble de micro-ordres", sans modifier l'architecture du compilateur. Si toutefois l'on veut garder une certaine universalité à la machine, et ne pas avoir une machine Algol 68, le choix des primitives reste difficile.

Signalons par ailleurs, l'existence de calculateurs dont la technologie est plus élaborée, tels que les machines MU5 [CAPO-MU5] ou BURROUGHS 6700 [ORGA-CLE], pour lesquels certains problèmes de traduction que nous évoquons ici, disparaissent. Dans ce dernier par exemple, l'utilisation systématique de "préfixes" pour chaque élément mémoire permet de ne pas engendrer d'instructions pour les indirections ou les conversions de type ; les opérations effectuées dépendent du type de ses opérandes. Dans ce cas, les quadruplets tels que nous les avons définis, s'adaptent bien aux instructions machine que l'on doit produire.

1.4 - Influence du calculateur sur la définition du compilateur -

Les incidences du calculateur sur le compilateur portent essentiellement sur la représentation des valeurs (définition pour les valeurs standard et optimisation pour les autres) et sur la traduction des quadruplets. Dans la description, donnée dans la deuxième partie, elles interviennent au niveau des processus de décorations de l'arbre abstrait, et de la génération de code.

- Les processus de décoration concernés dépendent de la gestion des objets utilisés dans le programme à compiler (zone des variables). Pratiquement, dans le compilateur, les décorations sont : soit représentées (voir ch. III.4) sous forme de tables, soit construites à partir de tables. Les particularités technologiques du calculateur n'interviennent alors que dans l'initialisation de la table des modes.

- L'architecture du processus de génération de code est telle que ces incidences sont prises en compte, pour la plupart, au niveau du traducteur. Seules quelques optimisations locales interviennent, soit au niveau de la gestion statique des valeurs (en pratique, seuls les descripteurs de valeur sont concernés), soit au niveau du parcours de génération dans l'arbre abstrait.

Remarque : Nous n'avons pas tenu compte dans ce qui précède, de particularités technologiques trop locales. Par exemple, l'adressage du calculateur IBM 360 limite à 4K octets la taille de la zone adressable par déplacement par rapport à un registre de base. Dans l'architecture que nous proposons, de telles restrictions ne sont prises en compte qu'au niveau de la traduction et du chargement (cf. paragraphe 4), sans conséquences d'ailleurs pour l'utilisateur.

2 - LES VALEURS ALGOL 68 ET LE COMPILATEUR -

Nous étudions maintenant quelques problèmes liés aux valeurs dans la réalisation du compilateur. La description, donnée dans la deuxième partie, a permis de mettre en valeur l'importance de ces problèmes. Nous montrons ici les aménagements réalisés par rapport à cette description. Ils correspondent à la dernière étape du raffinement, et sont souvent rendus possible par la technologie du calculateur pour lequel est écrit le compilateur.

Nous allons tout d'abord préciser la structure de l'environnement d'exécution des programmes Algol 68, en fixant les représentations des objets déclarés.

Nous envisageons ensuite quelques aspects de la génération de code, liés à l'utilisation dynamique des valeurs dont la représentation est arborescente. Une étude assez complète de ce problème important en Algol 68, où l'utilisateur peut être amené à manipuler des structures de données complexes, a été faite par l'équipe "Algol 68" du MBLE [BRANQU-1].

Nous proposons enfin quelques extensions aux descripteurs de valeur tels que nous les avons définis jusqu'ici.

2.1 - Représentations associées aux objets déclarés -

Dans la deuxième partie, nous avons donné (à propos de l'environnement à l'exécution) un aperçu fonctionnel de la représentation des valeurs. Nous allons préciser cette représentation en séparant les valeurs de mode tableau et repère de tableau, des autres.

2.1.1 - Représentation des valeurs -

Ces valeurs, pour être utilisables, doivent être possédées par des objets déclarés dans le programme.

Il est intéressant de remarquer que s'il n'existe, dans la définition du langage, qu'une seule manière de déclarer des indicateurs de mode, des indicateurs de priorité ou des opérateurs, il existe deux possiblités pour déclarer des identificateurs :

- déclaration de variable,
- déclaration d'identité.

Dans une première approche de l'implantation, il est normal de considérer une déclaration de variable comme étant une déclaration d'identité avec générateur.

Exemple : *ent x ;* <=> *rep ent x = loc ent ;*

Avec cette optique, la déclaration de variable n'est qu'une commoditié d'écriture fournie à l'utilisateur. Cependant, la syntaxe de la construction ainsi obtenue n'est pas aussi générale que celle d'une déclaration d'identité car la partie droite ainsi créée est toujours un générateur. D'autre part le mode associé est toujours un mode repère.

Grâce à ces informations statiques, les mécanismes et les solutions adoptés pour les représentations associées aux déclarations d'identité peuvent être rendus plus efficaces dans le cas particulier des déclarations de variable.

Soient dans la portée de la déclaration :

mode m = ...

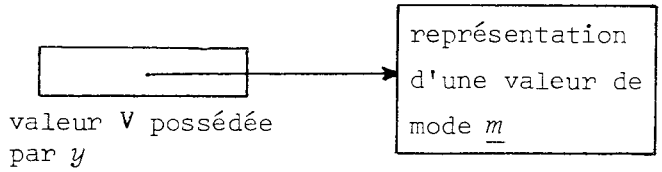
les deux déclarations suivantes :

m x ;
rep m y = ... ;

x et y ont tous les deux le mode *rep* m.
Etant donné que la représentation d'une valeur de mode repère est un pointeur, nous avons, pour la déclaration de y :

- une évaluation de la partie droite. Ceci fournit une valeur de mode *rep* m (et donc l'accès à la représentation d'une valeur de mode m)
- une mise en place de la relation "posséder" entre y et le pointeur délivré.

En définitive, nous obtenons la configuration suivante :

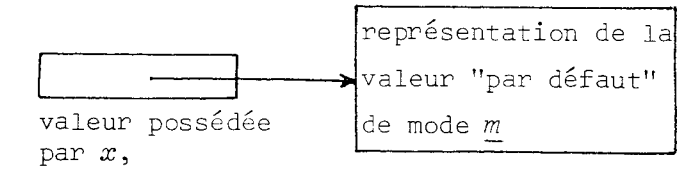


L'espace utilisé par la représentation de V (sous forme de pointeur) est alloué dans la zone des variables de l'environnement courant.

Pour la déclaration de x, nous avons,en respectant les mêmes conventions :

- une allocation et une structuration de l'espace associé au générateur sous-entendu (représentation de la "valeur par défaut" de mode m)
- une mise en place du pointeur sur cette représentation.

Ce qui donne finalement :



la représentation associée se trouve dans la zone des variables.

Examinons les conditions permettant de supprimer la représentation du pointeur, c'est-à-dire d'avoir, dans le cas d'une déclaration de variable, une représentation "dérepérée".

- Les problèmes de compatibilité des modes étant résolus statiquement, peu importe la représentation des valeurs associées.
- La valeur possédée par l'objet déclaré n'est plus la représentation de la valeur du pointeur (de mode *rep* *m*, mais l'accès à la représentation de la valeur pointée (de mode *m*). Etant donné qu'il s'agit d'une valeur manipulable par l'utilisateur, il faut disposer d'un calculateur ayant, au moins, une instruction permettant de traiter des adresses (instruction LA citée précédemment, par exemple).
- De manière à garder un accès statique à la valeur possédée par l'objet, la représentation statique de la valeur de mode *m* doit être allouée dans la zone des variables.
- Ce changement ne doit intervenir que sur la représentation associée à la variable déclarée. En conséquence, tout ou partie de la représentation à modifier ne doit pas être utilisée par d'autres constructions. Cette condition oblige à allouer et à structurer l'espace (pour la représentation de la valeur par défaut de mode *m*), de manière à avoir une représentation associée, uniquement, à la déclaration correspondante. C'est ce qui se passe pour les déclarations de variable, pour lesquelles le générateur est sous-entendu.

Moyennant le respect de ces conditions, nous pouvons associer à une variable, sa représentation dérepérée. Montrons que ceci ne peut pas être généralisé aux déclarations d'identité.

Les déclarations d'identité permettent de faire posséder une même valeur par plusieurs objets. La représentation associée est alors partagée. D'autre part, associer directement à l'identificateur déclaré (de mode *rep* *m*) la représentation de la valeur pointée (de mode *m*), ne permet plus d'avoir un accès statique à la valeur possédée par l'identificateur déclaré, puisque cet accès est fourni dynamiquement par l'évaluation de la partie droite. On pourrait envisager la création d'un nouvel exemplaire de la valeur pointée (dont la représentation statique serait mise dans la zone des variables), ce qui résoudrait ces problèmes, mais ne respecterait plus la sémantique des déclarations d'identité puisqu'il n'y aurait plus partage de la même représentation.

Ayant défini, ainsi, deux types de représentation, nous allons avoir deux types de code engendré suivant que l'identificateur, dont on cherche à manipuler la valeur associée, a été déclaré au moyen d'une déclaration de variable ou d'une déclaration d'identité.

Durant la phase de génération de code, nous allons indiquer cette information sur le descripteur représentant l'objet possédant la valeur à manipuler.

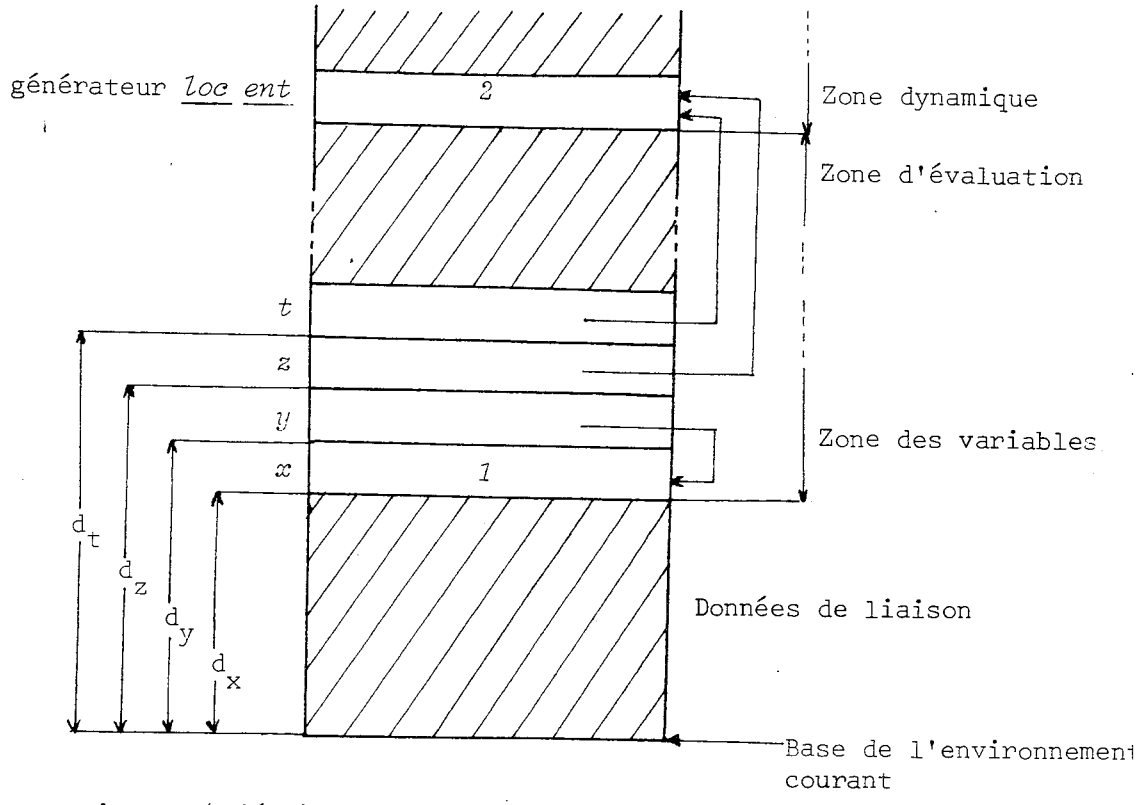
Nous parlons d'une représentation "stricte" dans le cas d'une déclaration d'identité, et "étendue" dans le cas d'une déclaration de variable.

Cette information va être utilisée chaque fois que l'on a besoin d'accéder à la représentation d'une valeur. C'est notamment le cas pour l'affectation, où il n'est pas nécessaire de produire de code pour une indirection sur la destination, lorsqu'elle est de représentation étendue.

Exemple :

```
        début
            ent x := 1
            rep ent y = x ;
            rep ent z = loc ent := 2 ;
            rep ent t = z ;
(A) ──→     ...
        fin
```

Lors de l'élaboration de ce programme, nous avons la configuration suivante, au point (A):



La représentation associée à x est une représentation étendue. Celles associées à y, z et t sont strictes.
La valeur possédée par x est l'adresse de l'emplacement se trouvant au déplacement d_x par rapport à la base de l'environnement courant. Les valeurs possédées par y, z et t sont les contenus des emplacements se trouvant respectivement aux déplacements d_y, d_z, d_t par rapport à la base de l'environnement courant.

2.1.2 -Un deuxième problème de représentation des valeurs concerne les valeurs de mode tableau et de mode repère de tableau.
Nous avons indiqué (voir ch. II.4.1, § 2.4) à propos du système à l'exécution, que la représentation d'une valeur_tableau était formée de deux parties :
- la représentation statique comprenant notamment:
  - les bornes inférieures et supérieures ainsi que les enjambées de chaque dimension,
  - l'origine virtuelle.
- la représentation de la partie dynamique formée de la représentation des éléments du tableau.

Il est nécessaire de préciser cette représentation.

- l'origine virtuelle est un pointeur, ce qui correspond plutôt à la représentation d'une valeur de mode repère. En effet, une valeur de mode tableau n'est formée que de l'ensemble des valeurs de ses éléments. Malheureusement, ce nombre d'éléments étant dynamique nous ne pouvons pas conserver un accès statique sur chacun d'entre eux. D'où, dans sa représentation, la présence d'une partie statique permettant d'accéder aux éléments. La transmission d'une telle valeur consiste donc à passer la représentation de ses éléments accompagnée (si besoin est) de la partie statique considérée uniquement comme moyen d'accès. On retrouve la notion de fonction d'accès introduite par Hoare [DAHL-HOA].

Exemples :

a) - dans l'affectation d'une valeur tableau, seuls les éléments sont recopiés, la représentation de tableau associée à la destination gardant son propre moyen d'accès aux représentations des éléments.

b) - dans le passage d'un paramètre de mode tableau, les représentations des éléments et le moyen d'y accéder (partie statique) sont transmis à la routine.

- l'impossibilité de fournir une représentation totalement statique pour une valeur de ce mode oblige, dans les déclarations d'identité, à allouer (en cours d'exécution) de l'espace pour la représentation des parties dynamiques de la valeur.

La représentation d'une valeur de mode repère de tableau se déduit des critiques faites sur celle des valeurs de mode tableau.

D'une manière générale, pour représenter une valeur de mode repère, on représente l'accès à la valeur repérée. La représentation d'une valeur repère de tableau est donc celle de l'accès aux éléments du tableau (puisque une valeur de mode tableau est formée de l'ensemble des valeurs de ses éléments). D'après ce que nous venons de voir, la représentation statique d'une valeur tableau convient parfaitement pour représenter une valeur repère de tableau. La transmission d'une telle valeur se résume à celle de la représentation statique de la valeur tableau associée.

Exemples :

a) - dans l'affectation d'une valeur repère de tableau, seul le moyen d'accéder aux éléments est recopié.

b) - dans le passage d'un paramètre repère de tableau, seul le moyen d'accéder aux éléments est transmis à la routine à appeler.

Remarques :

1) - Une telle représentation pour une valeur de mode repère de tableau résout les problèmes de portée posés par les opérations de "tranche" effectuées sur cette valeur [BRANQU-3]. Effectuer une tranche, consiste simplement à créer une nouvelle fonction d'accès à partir de celle qui est fournie. Par contre, faire une tranche sur une valeur _ tableau nécessite (en principe) de créer (par copie) la représentation des éléments de la nouvelle valeur_tableau et de fournir la fonction d'accès correspondante. La création de la représentation des éléments peut être évitée en tenant compte des remarques faites sur l'évaluation collatérale (voir ch. II.4.1, § 2.2).

2) - La représentation d'une valeur de mode repère de tableau flexible ne peut pas être semblable à celle que nous venons d'indiquer pour les repères de tableaux fixes. Le nombre d'éléments de la valeur tableau accessible par un repère de tableau flexible varie au cours de l'évaluation d'un programme. Dans ces conditions toute affectation à un repère de tableau flexible nécessite une allocation d'espace pour la représentation des nouveaux éléments du tableau, et la modification du moyen d'accès à ces éléments, ce qui n'est pas le cas pour l'affectation à un repère de tableau fixe.

L'utilisateur peut définir plusieurs valeurs_repère_de_tableau_flexible qui pendant l'exécution de son programme permettent d'accéder à la même représentation d'une valeur_tableau. Comme nous venons de le voir, la fonction d'accès étant susceptible de modifications, elle ne peut plus être associée à chaque valeur_repère et doit donc être commune à l'ensemble de ces valeurs. Afin de respecter cette condition, la représentation d'une valeur_repère_de_tableau_flexible est formée d'un pointeur sur celle de la valeur_tableau. Une telle représentation est bien adaptée car le langage ne permet pas de manipuler des valeurs de mode repère de tableau flexible obtenues au moyen de tranches.

2.2 - Génération de code pour l'utilisation des valeurs -

Dans la description (voir ch. II.4) nous avons précisé le processus de génération de code à partir d'un arbre abstrait. Bien que cette description fasse apparaître le code permettant d'accéder à la représentation d'une valeur, elle

n'indique pas celui qu'il faut engendrer pour utiliser cette valeur. Etant donné qu'un langage est fait pour permettre la manipulation de valeurs il ne suffit pas de savoir produire l'accès à une valeur, encore faut-il être capable d'engendrer le code réalisant, sur celle-ci , l'action demandée.

2.2.1 - Rappels sur la représentation des valeurs -

Nous avons indiqué précédemment la représentation des différents types de valeurs Algol 68 (voir § 2.1.2 de ce chapitre et surtout ch. II.4.1, § 2.4). Rappelons quelques unes de ces représentations :

- la représentation d'une valeur de type structure est formée de l'ensemble des représentations de ses champs,
- la représentation d'une valeur de type union est formée d'un en-tête (décrivant le mode, à l'exécution, de la valeur courante) suivi de la représentation de la valeur courante.
- les autres valeurs ont des représentations de taille fixe et indépendantes des sous-modes apparaissant dans leur déclareur de mode.

  Ex : deux mots pour une routine,
  un mot pour un pointeur.

En respectant les règles de construction des modes nous en déduisons que seules les valeurs de type tableau, structure ou union peuvent avoir des représentations formées d'une partie statique et d'une partie dynamique. Signalons d'autre part que les valeurs sans parties dynamiques - et les parties statiques des valeurs à parties dynamiques - ont toutes des représentations continues. Il n'en est pas de même avec les représentations des parties dynamiques. En effet l'opération de tranche sur un tableau ou de sélection multiple sur un tableau de structures conduit à avoir des représentations non continues pour les parties dynamiques des valeurs ainsi obtenues.

2.2.2 - Génération en fonction de la représentation des valeurs -

Reprenons le problème de la génération du code utilisant une valeur pour une action donnée.

Dans le cas des valeurs sans parties dynamiques, la représentation étant continue, et de taille connue à la compilation, la génération ne pose pas de problèmes, le code ne dépendant que de l'action à réaliser. En pratique,

suivant la diversité des instructions utilisables (sur le calculateur), un certain nombre de cas peuvent être différenciés - en fonction du type de la valeur - pour produire un code optimal.

Le problème n'est pas aussi simple lorsqu'il s'agit de valeurs ayant des parties dynamiques. Nous disons que de telles valeurs ont des représentations arborescentes. L'utilisation d'une de ces valeurs nécessite de parcourir sa représentation afin d'accéder aux valeurs composant les parties dynamiques. Indépendamment de toutes les optimisations qu'il est possible de réaliser lorsque les représentations des parties dynamiques sont continues, le code utilisant la partie dynamique d'une valeur réalise le traitement suivant :

pour chaque valeur composant la partie dynamique :

- accès à la représentation de cette valeur,
- manipulation de cette valeur.

Si cette valeur est, elle aussi, à parties dynamiques, l'utilisation de chacune d'entre elles suit le même principe. L'utilisation d'une valeur à parties dynamiques nécessite donc la génération du code permettant de parcourir sa représentation arborescente jusqu'à accéder aux représentations des valeurs - sans parties dynamiques - qui la composent.

Du point de vue descriptif, il n'y a pas de raison de traiter différemment les valeurs suivant qu'elles présentent ou non des parties dynamiques. En effet, sur une machine (abstraite) idéale, la création et l'utilisation de valeurs se réaliseraient toujours de la même manière quelles que soient leurs caractéristiques. C'est pourquoi la description que nous avons donnée précédemment ne fait pas intervenir les problèmes de parcours des représentations arborescentes des valeurs. D'autre part, la génération de code pour ces parcours est fonction de la représentation adoptée pour les valeurs. Le code produit pour le parcours de la représentation d'une valeur arborescente est indépendant de l'action à réaliser sur celle-ci. Il ne dépend que de son mode.

2.2.3 - Génération de code pour le parcours de la représentation arborescente d'une valeur -

Dans ce paragraphe nous indiquons le traitement effectué pendant la phase de génération pour obtenir le code permettant l'accès à chaque valeur composante d'une valeur à parties dynamiques. Nous utilisons le même formalisme

que dans la deuxième partie (voir ch. II.4). D'après l'analyse précédente nous distinguons trois cas :

- les tableaux qui sont des valeurs à parties dynamiques,
  les structures dont les champs peuvent avoir des parties dynamiques,
  les unions dont la valeur courante (à l'exécution) peut être une valeur à parties dynamiques.

Dans chacun de ces cas nous avons un descripteur δval décrivant l'objet possédant la valeur (à parties dynamiques) à parcourir.

```
action manipulation = (descr δval)
    cas type modedescr(δval)
    dans
        co génération suivant le mode de la valeur à parcourir co
      Structures :
        co indépendamment de la manipulation à faire, le code engendré
           pour la structure est la juxtaposition du code produit pour
           chacun de ses champs co
           descr δchamp
           pourtout champ champstruct ∈ co la structure décrite par δval co
           faire
              δchamp ← modifieraccès(déplacement(champstruct), δval)
              δchamp ← complétermode(δchamp, modech(champstruct))
              manipulation(δchamp)
                 co manipulation du champ avec génération de code pour le
                    parcours (si besoin est) co
           finfaire
              co ceci revient à produire, comme code de parcours, le code
                 de sélection (simple) de chaque champ de la structure co
```

Tableaux :

co Production du code permettant l'accès à tous les éléments du tableau. Une possibilité est de considérer que la manipulation du tableau est équivalente à celle (par indexation) de chaque élément du tableau. Nous pouvons alors engendrer un nombre de boucles imbriquées égal au nombre de dimensions du tableau, la variable contrôlée de chaque boucle prenant (dans un ordre quelconque) toutes les valeurs entières comprises entre celle de la borne inférieure (incluse) et celle de la borne supérieure (incluse) de la dimension correspondante. A l'intérieur de la boucle la plus interne, l'indexation du tableau avec, comme indices, les valeurs des variables contrôlées, permet d'accéder à l'élément correspondant.

Ce code correspond à celui qui est produit lorsque l'utilisateur écrit la manipulation de tableau élément par élément. Son principal défaut est de nécessiter autant d'indexations que le tableau a d'éléments.

Il est bien sûr possible d'améliorer ce code afin d'éviter toutes ces indexations [68TECH].

co

Unions :

co Indépendamment de la manipulation de l'en-tête, il faut, pour chaque sous-mode composant de l'union, produire le code manipulant une valeur de ce sous-mode. Ce code est comparable à celui produit pour le choix union (voir ch. II.4.6, § 1.1.2), la seule différence étant qu'il effectue le traitement pour tous les modes possibles de l'union.

A l'exécution, le mode de la valeur courante permet (par indexation de la table des alternances) de récupérer le point d'entrée de la séquence de code correspondante. Cette séquence de code est alors exécutée. Elle se termine par un branchement à l'étiquette placée après toutes ces séquences co

descr δsousmode

co *génération du code traitant l'en-tête* *co*

co *génération du code indexant la table des alternances et réalisant le branchement à l'étiquette ainsi sélectionnée* *co*

pourtout type $sousmode_i$ ∈ *co ensemble des modes composants de l'union représentée par* δval *co*

faire

co *génération de l'étiquette* E_i *co*

sousmode ← δval

sousmode ← *complétermode*(δsousmode, $sousmode_i$)

manipulation(δsousmode)

co *manipulation de la valeur courante avec génération de code pour le parcours, si besoin est* *co*

co *génération du branchement à l'étiquette de fin* *co*

finfaire

co *génération de l'étiquette de fin* *co*

finaction

2.2.4 - Utilisation des parcours d'arborescence -

Ces parcours sont produits pendant la génération de code pour les constructions suivantes :

- Création par l'utilisateur de valeurs à parties dynamiques -

Cette création peut se faire de deux manières :

. au moyen d'un générateur :

explicite, dans les noeuds :

(s-générateur) , (d-générateur) , (i-générateur)

implicite dans les noeuds :

(s-variable) , (d-variable) , (i-variable)

. au moyen d'une déclaration d'identité

(déclaration-identité)

- Transmission, avec changement d'environnement, de valeurs à parties dynamiques. Il s'agit des passages des paramètres (noeud (routine)) aux routines.

- Affectation de valeurs à parties dynamiques -
C'est le cas du noeud (affectation)

- Protection de valeurs à parties dynamiques -
Les valeurs sont à protéger (par création d'un nouvel exemplaire) lorsqu'elles sont en position collatérale. Cela intervient dans les noeuds :

(affectation) , (appel) , (opérateur) ,

(collatéral) , (tranche) , (index) .

- Lecture et écriture de valeurs -
Il s'agit du noeud (entrée-sortie) dont le traitement nécessite la génération du code "déployant", en valeurs simples, toutes les valeurs à lire ou à écrire, qu'elles aient ou non des parties dynamiques.

Si pour les quatre premiers cas, les champs d'une structure ou les éléments d'un tableau peuvent être parcourus dans un ordre quelconque, sans changer la sémantique du programme source, il n'en est pas de même pour l'opération d'entrée-sortie. En effet, que dirait l'utilisateur, si les champs des structures ou les éléments des tableaux qu'il désire lire ou écrire, étaient traités dans un ordre aléatoire ?

2.3 - Les descripteurs de valeurs -

Nous venons d'introduire dans cette partie de nouvelles informations décrivant une représentation plus optimale des valeurs. Elles se retrouvent dans le descripteur associé à chaque valeur. En plus des informations fonctionnelles introduites dans la deuxième partie, nous indiquons donc dans le descripteur :

a) - le type de la représentation de la valeur, pour prendre en compte les optimisations utilisant les opérandes immédiats, le code condition, etc....
Le descripteur ne comporte pas de base ni de déplacement, mais la valeur immédiate ou le masque associé au code condition.

b) - le type de la ressource (mémoire ou registre) dans laquelle se trouve la valeur décrite.

c) - une indication permettant de distinguer les représentations d'objets de type strict ou étendu (voir § 2.1.1 de ce chapitre). Nous envisageons maintenant les divers traitements effectués sur ces descripteurs pendant la génération de code.

2.3.1 - Gestion des descripteurs -

Les routines de création de descripteurs à partir des décorations trouvent une partie des informations nécessaires dans les diverses tables.

Il nous faut maintenant donner une représentation des mécanismes d'allocation-libération et de synthèse -héritage des chaînes de descripteurs. Une solution simple est d'utiliser pour le rangement de ces chaînes une pile associée à la fonction de parcours de l'arbre abstrait. Remarquons qu'il existe une corrélation entre cette pile et la pile d'évaluation.

Pour chaque noeud parcouru, on crée au sommet de la pile des descripteurs, un nouvel environnement dans lequel sont alloués tous les descripteurs déclarés localement dans la description fonctionnelle, et plus généralement tous les descripteurs utilisés temporairement pendant la génération de code. Lorsque cet environnement est créé, il comporte éventuellement, les descripteurs hérités sur le noeud; lorsqu'il est libéré, il est remplacé en sommet de pile par le descripteur synthétisé sur le noeud. On peut ainsi facilement représenter un ensemble ordonné de descripteurs synthétisés tel qu'il est utilisé dans le noeud (index) par exemple.

Les descripteurs étant constitués de chaînes d'objets de même format, il est commode d'associer à la pile une liste libre. Seules les "têtes de chaîne" sont allouées sur la pile, les descripteurs constituant le reste de la chaîne sont eux, alloués dans la liste libre. On alloue également, en utilisant cette liste libre, les ensembles de descripteurs globaux sur lesquels on garde un accès permanent (par exemple descripteurs représentant le vecteur des accès aux objets non locaux). La libération d'un tel environnement nécessite en général le parcours d'une ou plusieurs chaînes et(ou) listes, et donc une restitution d'emplacements à la liste libre. Il est à noter que cette libération ne se limite pas à une simple gestion de descripteurs ; on libère également les ressources (en particulier les registres) éventuellement allouées pour les valeurs décrites par ces descripteurs.

2.3.2 - Modifications et descripteurs de valeurs -

Rappelons qu'un des rôles des descripteurs est de permettre la production de code le plus tard possible en transmettant toutes les informations nécessaires à cette génération lors des synthèses.

Nous avons montré que les transformations statiques de valeurs étaient effectuées sur le descripteur. Par exemple, la modification dérepérer transforme la chaîne de descripteurs associée à la valeur dérepérer. La fonction d'accès du descripteur est modifiée comme suit :

- si la représentation de la valeur est stricte, on crée une indirection en ajoutant un élément en tête de la liste représentant le descripteur.
- si la représentation de la valeur est étendue on transforme, dans le descripteur, l'indicateur étendu en indicateur strict.

Nous allons traiter d'une manière analogue les modifications unir et ranger transformant la classe de la valeur. Il n'est en effet pas nécessaire de créer une représentation de la valeur modifiée, il suffit d'indiquer sur le descripteur de la valeur primitive la modification en précisant son type et les modes a priori et a posteriori qui lui sont associés.

Exemple :
Soit le programme Algol 68

```
début
    ent a = 1, union (ent, réel) b ;
    b := a
fin
```

la valeur possédée par *a* de mode entier est unie à une valeur de mode *union (ent, réel)*. L'élaboration de cette modification conduit à créer une nouvelle représentation pour cette valeur unie dans la zone d'évaluation, par exemple. L'élaboration de l'affectation qui suit, va affecter cette valeur unie. La représentation de la valeur, primitivement possédée par *a* , sous forme d'une valeur de mode union, n'est pas utile; il suffit d'indiquer la modification dans le descripteur associé et d'en tenir compte lors de la génération de code pour l'affectation (affectation d'une valeur entière dans une variable de mode *rep union (ent, réel)*). Par ce moyen nous évitons de créer dynamiquement une représentation fugitive pour une valeur. Le gain

obtenu ainsi à l'exécution entraîne une complexité plus grande des routines de génération de code pour l'affectation.

Nous avons ainsi précisé l'ensemble des informations contenues dans les descripteurs et les principes de gestion de ces descripteurs. On trouve dans les notices techniques une description complète des solutions utilisées.

3 - CALCUL COLLATERAL ET OPTIMISATIONS -

Il ne s'agit pas de décrire des techniques d'optimisations globales dont Exel présente dans sa thèse [EXEL] un panorama assez complet. Le compilateur ne comporte pas de phase d'optimisation des programmes Algol 68 compilés. Elle peut s'insérer dans le compilateur avant la génération de code et s'effectuer à partir de l'arbre abstrait décoré (qui tient lieu de texte intermédiaire). Nous ne nous intéressons ici qu'aux possibilités (offertes par le calcul collatéral) de changer l'ordre d'élaboration. Après avoir rappelé les différents parcours de génération nous proposons quelques optimisations dans la gestion dynamique des valeurs.

3.1 - Parcours de génération dans l'arbre abstrait -

Nous avons introduit plusieurs types de parcours d'arbre pour la génération de code. Ils sont notés différemment dans la description. Nous allons préciser une implantation possible de ces divers parcours. Pour effectuer éventuellement des optimisations, l'implantation peut tenir compte de la signification de chaque parcours et des possibilités technologiques de calcul simultané du calculateur.

3.1.1 - Implantation des parcours -

Le parcours, schématisé par : *séquence*

finséquence ,

permet de produire le code pour une élaboration strictement séquentielle. Nous l'avons utilisé dans le noeud (entrée-sortie) par exemple. On peut l'implanter simplement par le parcours canonique habituellement défini sur les arborescences. Les parcours pour lesquels aucun ordre n'est fixé sont

de deux types : soit on peut leur associer une signification sémantique, nous les qualifions alors de parcours collatéraux et nous envisageons des optimisations pour l'élaboration du programme, soit ils n'ont pas de signification particulière et les optimisations peuvent être reportées au niveau du compilateur lui-même.

Dans le second cas, nous l'avons schématisé par :

pour tout arbre $x \in$ *ensemble arbre* A
faire
.
.
.
finfaire

Il s'agit d'un parcours d'énumération d'un ensemble, en général d'arborescences, tel que chaque élément soit parcouru sans que l'ordre d'énumération soit précisé. Aucune signification sémantique ne lui étant attachée, aucune information n'est synthétisée sur la racine dans le cas des arborescences. Ce parcours exprime seulement l'indépendance de la génération de code pour chacune de ces arborescences. L'absence de signification sémantique est évidente dans les cas où ce parcours est utilisé.

Exemples :
- liste de choix : à l'exécution une seule séquence de code est évaluée.
- liste de déclarations associées à un déclareur : toutes les évaluations sont indépendantes et ne délivrent aucune valeur_résultat.

Si la machine sur laquelle s'exécute la compilation dispose de possibilités de calcul parallèle, on peut éventuellement, envisager de produire simultanément les séquences de code associées aux arborescences. La mise en évidence d'un tel parcours a surtout un intérêt descriptif. En général, on transforme ce parcours en parcours canonique comme dans le cas séquentiel.

L'autre type de parcours correspond à une évaluation collatérale. Il exprime, comme précédemment, l'indépendance dans l'ordre de génération des séquences de code. Toutefois, à l'exécution l'ensemble des valeurs_résultat , délivrées par ces séquences, est nécessaire à la poursuite de l'élaboration ; elles se retrouvent toutes simultanément dans l'environnement courant. Par exemple, pour l'indexation les valeurs des indices et du primaire sont nécessaires pour effectuer l'opération.

Ce parcours est schématisé par :

collatéral
.
.
.
fincollatéral

Nous présentons quelques possibilités d'implantation offertes par ce type de parcours.

3.2 - Evaluation collatérale et optimisation -

Nous avons suggéré précédemment, une utilisation des possibilités technologiques de calculs simultanés du calculateur dans lequel est évalué le programme, pour optimiser sous certaines conditions cette évaluation. Il est évident que s'il ne dispose pas de ces possibilités on peut imposer un ordre séquentiel d'évaluation, le parcours de génération devenant alors canonique.

Nous envisageons certaines optimisations dans l'implantation, rendues possibles par le langage Algol 68 (indépendamment du calculateur). Elles permettent d'utiliser la collatéralité de certains noeuds de l'arbre abstrait lors de l'élaboration du programme, donc lors du parcours de génération. Les optimisations possibles sont nombreuses ; nous pouvons les classer en deux groupes :

- les optimisations statiques portant essentiellement sur la gestion des ressources privilégiées (registres) dont le nombre est limité,

- les optimisations, que nous qualifions de dynamiques, qui permettent de préserver l'accessibilité aux valeurs en évitant des copies. Il faut noter que ces dernières constituent des entorses à la définition sémantique stricte du langage, sans conséquence pour le résultat de l'élaboration.

- Optimisations statiques -

A titre d'exemple nous proposons une optimisation possible. Une expression, bien que représentée dans l'arbre abstrait sous forme d'un arbre binaire, se comporte comme un ensemble d'unités que l'on peut évaluer collatéralement. La même remarque s'applique à toute liste d'opérandes (dans l'arbre abstrait) traitée collatéralement (par exemple, dans les noeuds (appel), (index), etc...).

Pour fixer un ordre d'évaluation on peut choisir un critère portant sur l'allocation des registres. On peut adapter, par exemple, l'algorithme de Sethi-Ullman [SEHTI]. Cet algorithme permet de déterminer le nombre de registres nécessaires à l'évaluation d'une expression ; on peut par suite, en fonction de cette information, fixer un ordre d'évaluation de l'expression. L'évaluation collatérale généralise l'évaluation d'une expression et, de la même manière, il est possible avec cet algorithme d'optimiser l'utilisation des registres. On impose ainsi un ordre d'évaluation et, par suite, un ordre dans le parcours de génération.

Remarque : Nous avons vu que le traitement d'un appel nécessite de disposer de tous les registres. Avec cette stratégie, si un appel apparait en position collatérale, il est effectué en premier.

- Optimisations dynamiques -

En ne respectant pas strictement la définition de la sémantique du langage, on peut sans changer la signification d'un programme éviter certaines copies de valeurs. Nous sommes amenés à réexaminer l'implantation du système à l'exécution définie dans la deuxième partie (voir ch. II.4.1, § 2).

a) - Gestion de la pile à l'exécution -

Considérons l'exemple suivant :

```
début
    ent a, b, c ; ... ; co environnement principal co
    proc p := [ ] ent    : fant ;
    ...
    (p co l'élaboration du premier constituant délivre le tableau t1 co,
     p co l'élaboration du deuxième constituant délivre le tableau t2 co
     p co l'élaboration du troisième constituant délivre le tableau t3 c
fin
```

Nous examinons pour cette phrase collatérale, le cas d'une évaluation séquentielle et le cas d'une évaluation parallèle de chacun des trois constituants.

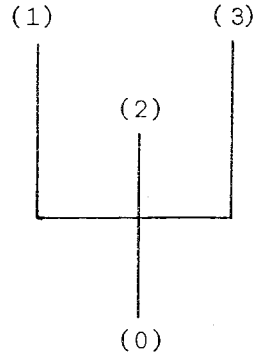
- Evaluation séquentielle -

Les trois composants sont évalués les uns après les autres. Nous fixons arbitrairement l'ordre séquentiel de gauche à droite. Les parties statiques des

tableaux se trouvent dans la zone des variables, appartenant à la partie statique de la pile ; les éléments des tableaux appartiennent à la partie dynamique de la pile. Au cours de l'évaluation, les parties statiques de la pile vont se recouvrir, les parties dynamiques des tableaux t1 et t2 peuvent être détruites(voir ch. II.4.1 § 2). Il est nécessaire si l'on veut garder les valeurs t1, t2, t3,de copier les parties statiques de leurs représentations dans la zone d'évaluation de l'environnement principal ; les parties dynamiques le sont avant chaque nouvelle allocation d'espace risquant de les détruire. On copie ainsi, éventuellement, les parties dynamiques des tableaux t1 et t2. (Rappelons que l'espace alloué pour les parties dynamiques n'est récupéré qu'en fin d'élaboration de routine).

- Evaluation parallèle -

L'évaluation en parallèle des trois constituants de la phrase collatérale s'effectue par l'intermédiaire d'une structure de données que l'on a pris l'habitude de qualifier de pile cactus [VERJUS]. Dans le cas de notre exemple,l'évaluation des trois appels de *p* va s'effectuer indépendamment et mettre en oeuvre trois piles suivant le schéma :



(0) : représentation de l'environnement principal

(1), (2), (3) : représentation de la zone allouée pour l'évaluation de chaque appel de la procédure p

Remarques :

1) - L'environnement d'évaluation de chaque appel est formé de la réunion de l'environnement principal (0) et de la zone allouée (1), ou (2), ou (3). Une description plus complète des structures mises en oeuvre pour une telle évaluation est faite dans l'étude de l'implantation du calcul parallèle [68GRE-8].

2) - La zone allouée est formée d'une zone des variables et d'évaluation dont la taille est connue statiquement et d'une zone dynamique contenant les éléments des tableaux t1, t2 ou t3 dans l'exemple.

Dans cette évaluation les zones statiques de la pile d'évaluation ne se recouvrent plus et les parties statiques ne risquent plus d'être détruites par une allocation d'espace. Ainsi la dissociation des trois évaluations évite la copie systématique des parties statiques et éventuellement des parties dynamiques, dans le seul but de préserver l'accessibilité à la valeur ou de ne pas perdre cette valeur. (Dans le contexte de l'évaluation séquentielle précédente, les copies des parties statiques de t1 et t2, les copies éventuelles des éléments de t1 et t2 seraient supprimées).

Nous pouvons remarquer qu'une telle évaluation n'évite pas les copies provoquées par la transmission d'une valeur_résultat lors d'un changement d'environnement.

Nous allons essayer, en reprenant l'évaluation séquentielle qui reste la seule possible sur toute une classe de calculateurs, de nous rapprocher des conditions favorables (pour les copies de valeurs) que l'on trouve avec l'évaluation parallèle. Nous pouvons, en effet, ne plus fixer arbitrairement l'ordre d'évaluation (de gauche à droite) pour les constituants d'une phrase collatérale mais choisir pour cet ordre le critère permettant de minimiser le nombre de copies.

b) - Stratégie d'évaluation collatérale -

Dans une évaluation collatérale les parties statiques des valeurs en position collatérale sont toujours rendues accessibles si l'on n'effectue plus de recouvrement des zones statiques (voir paragraphe précédent).

Dans le schéma d'évaluation séquentiel que nous avons rappelé nous sommes amenés, pour une valeur à partie dynamique en position collatérale, à effectuer une copie pour préserver son accessibilité. Pour éviter cette copie systématique on peut laisser en place, dans certains cas, la partie dynamique de la représentation de la valeur. Elle peut cependant se trouver au dessus du sommet de la pile et par suite être détruite lors d'une allocation sur cette pile. L'allocation d'espace peut être nécessitée par l'évaluation d'un générateur ou par l'évaluation d'un appel de procédure (création d'un nouvel environnement). Nous allons examiner tous les cas possibles, à partir de l'arbre abstrait décoré.

L'ensemble des noeuds pour lesquels des valeurs à partie dynamique sont manipulées en position collatérale, est constitué des noeuds suivants:

(collatéral) , (appel) , (opérateur) , (identité) ,

(affectation) , (index) et (tranche) .

Un certain nombre de ces noeuds manipulent collatéralement plusieurs valeurs dont une seule est à partie dynamique, sans effectuer aucune allocation d'espace : il s'agit des noeuds (index) , (tranche) , et (affectation) . Nous excluons pour l'affectation le cas des tableaux flexibles. Pour ces noeuds nous pouvons élaborer la valeur à partie dynamique en dernier (primaire d'index ou de tranche, source de l'affectation) de telle sorte que son accessibilité soit conservée pendant le traitement de ces valeurs. On a ainsi imposé un ordre partiel pour le parcours collatéral de génération. Le traitement de tous les autres noeuds (sauf le noeud collatéral) nécessite une allocation d'espace, la partie dynamique doit être copiée pour préserver l'accessibilité à la valeur.

Dans le cas du noeud (collatéral) les valeurs en position collatérale peuvent être à partie dynamique et leur évaluation peut nécessiter une allocation d'espace.

Remarque :
Dans ces conditions plusieurs techniques sont envisageables pour effectuer la copie. On peut garder trace statiquement des valeurs qui se trouvent au-dessus du sommet de pile et les copier avant d'effectuer une allocation d'espace ; on peut ne copier que si l'allocation détruit une partie de la représentation de ces valeurs. Une autre possibilité est de transmettre, statiquement, une demande de copie conditionnelle pour la génération du code, et de permettre l'évaluation des valeurs en position collatérale. Lorsqu'une de ces valeurs se trouve au-dessus du sommet de pile (éventualité produite par un appel) elle est copiée si cette demande est effective. D'autres techniques peuvent être envisagées.

Toutes ces stratégies de gestion des valeurs dynamiques peuvent paraître compliquées; elles permettent toutefois d'éviter des copies systématiques de valeurs dynamiques, particulièrement coûteuses. La possibilité, offerte par le langage, d'utiliser en toutes circonstances ces valeurs dynamiques,peut devenir d'un coût excessif. Il paraît donc souhaitable, au niveau de l'implantation, d'essayer de réduire ce coût en limitant le nombre de copies, sans risque pour l'utilisateur ! (la définition du langage prévoit des créations de "nouveaux exemplaires de valeur").

4 - LES TRADUCTEURS -

Nous présentons ici les principes qui ont conduit à la conception des traducteurs. Rappelons que ces derniers sont au nombre de deux :

- le premier permet de passer des quadruplets au code objet ; il est qualifié de "codeur" dans les notices techniques, et prend en compte les particularités des instructions du calculateur pour lequel est produit ce code,

- le second permet de charger le code ; en plus de la résolution des problèmes de référence et d'adressage habituels pour un chargeur, nous pouvons lui adjoindre d'une part, certaines fonctions d'aide à la mise au point particulièrement intéressantes dans le contexte conversationnel où nous avons placé notre compilateur, d'autre part certaines possibilités d'édition de liens particulières à Algol 68.

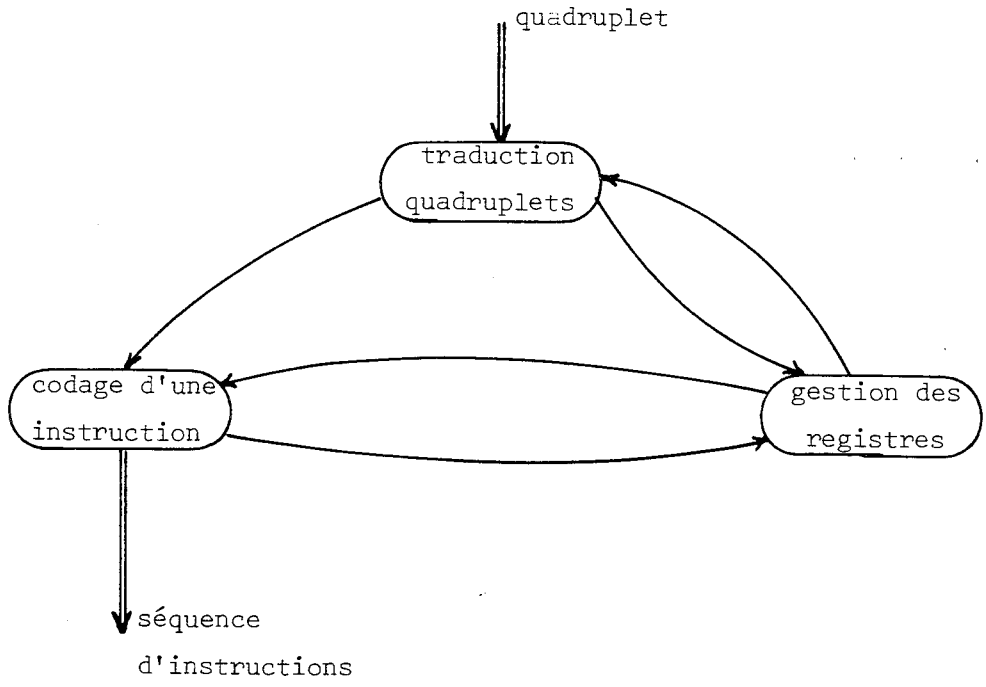
4.1 - Transformation des quadruplets -

La transformation des quadruplets en séquence d'instructions nécessite :

- de gérer les accumulateurs (registres dans notre cas) qui interviennent comme opérandes de ces instructions,

- de coder les instructions pour un calculateur donné à partir des chaînes de descripteurs opérandes du quadruplet,

- de définir, en fonction de la technologie du calculateur la séquence des instructions représentant le quadruplet. Nous avons ainsi déterminé les trois processus à partir desquels s'effectue la traduction. Nous pouvons schématiser leurs interactions.



Chacun d'eux peut être conçu indépendamment, une fois précisés leurs interfaces.

4.1.1 - Codage des instructions -

Le format des instructions à engendrer dépend de la technologie du calculateur. Le codage dans le format adéquat s'effectue à partir d'un code instruction et d'autant de chaînes de descripteurs de valeurs que l'instruction a d'opérandes. Ces descripteurs permettent de construire l'adressage des opérandes (registres ou mémoire), ou la valeur "immédiate" (pour un opérande immédiat). Dans le cas d'un opérande en mémoire le parcours d'une chaîne de descripteurs nécessite la construction d'indirections. Toutes les instructions du calculateur peuvent être produites, la plupart d'entre elles (à l'exception des instructions d'entrées-sorties et d'édition) sont d'ailleurs utilisées pour traduire les quadruplets.

Chaque instruction produite est précédée d'une requête de chargement. Elle indique au chargeur, le type du traitement à effectuer sur l'instruction et sur chacun de ses opérandes, en particulier.

Exemple : Les étiquettes utilisées dans le code objet, sont représentées par un entier, affecté pendant la génération. Lorsque l'étiquette est utilisée comme opérande d'une instruction (branchement par exemple) son numéro est placé dans le champ adéquat de l'instruction, et le type de l'opérande (utilisation d'étiquette) est précisé dans la requête de chargement associée.

A partir de ces informations le chargeur transforme l'étiquette en un déplacement par rapport à la base courante du code.

Nous précisons maintenant, dans le cas du calculateur IBM 360 les principaux problèmes rencontrés dans le codage des instructions :

a) - Les indirections n'étant pas codées directement dans l'adressage, le parcours des chaînes de descripteurs entraîne la génération répétée d'instructions de chargement d'un registre alloué à cette fin.

b) - Ce calculateur dispose de plusieurs formats d'instructions, ce qui complique le codage et a entraîné l'introduction d'un mécanisme de transformation automatique, permettant, suivant la nature des opérandes, de changer de format au moment du codage.

Exemple :
Le codage d'une instruction A (addition du contenu d'une mémoire à celui d'un registre général), avec deux descripteurs de registres comme opérandes, conduit à une instruction AR.

c) - Les problèmes concernant l'adressage des blocs mémoires (la zone mémoire adressable à partir d'une même base ne peut avoir une taille supérieure à 4K). ont été résolus au niveau du chargement.

Les opérandes des instructions sont codés sous forme d'un déplacement (sans restriction sur sa valeur) rangé dans le champ correspondant de l'instruction et d'un indicateur dans la requête de chargement associée. Cet indicateur précise la zone d'implantation de cet opérande, dans l'environnement d'évaluation. Les zones ainsi spécifiées sont, par exemple, la base courante de pile, la base de l'environnement du prélude standard, etc... .

A partir de ces informations le chargeur peut désigner une base pour l'opérande, après avoir inséré les instructions permettant éventuellement de ramener à 4K la valeur des déplacements.

Remarquons qu'aucune restriction n'est imposée à l'utilisateur, contrairement à certaines solutions plus simples, adoptées pour d'autres compilateurs limitant la taille des routines à compiler. Cette restriction n'est pas acceptable pour un compilateur destiné à compiler de gros programmes.

4.1.2 - Gestion des registres

4.1.2.1 - La gestion des registres (processus indépendant des deux autres) peut être la même pour un groupe de calculateurs disposant d'un ensemble semblable de registres. Les demandes d'allocation et de libération des registres sont effectuées par le processus de codage (pour engendrer des indirections par exemple) et de traduction (registres contenant un résultat intermédiaire).

Exemple :
Soit le quadruplet (+ :=, δ1, δ2,)
où δ1 est le descripteur d'une valeur entière en mémoire,

$(C_{ent}, (d_{\delta 1}, \lfloor \quad))$
$\lfloor \rightarrow$ δbase de l'environnement courant

δ2 est le descripteur d'une valeur entière en mémoire,

$(C_{ent}, (d_{\delta 2}, \lfloor \quad))$
$\lfloor \rightarrow$ δbase de l'environnement courant.

Sur le calculateur IBM 360 ce quadruplet est traduit par la séquence d'instructions :

$$\begin{aligned} &L \quad R, \quad d_{\delta 1} (R_{bc}) \\ &A \quad R, \quad d_{\delta 2} (R_{bc}) \\ &ST \quad R, \quad d_{\delta 1} (R_{bc}) \ , \end{aligned}$$

où R est un registre dont l'allocation a été demandée par le processus de traduction, et R_{bc} est le registre de base de l'environnement courant.

Le registre R est libéré après la traduction du quadruplet, il n'est utilisé que localement pour l'évaluation.

1) - On distingue plusieurs classes de registres :

- les registres servant de base dans l'environnement d'évaluation, et alloués de manière permanente (par exemple, base courante de pile, base du code, etc...).

- les registres de "service" utilisés par exemple pour communiquer avec les routines bibliothèque, alloués également de manière permanente.

- les registres utilisés pour la zone d'évaluation. Pour ceux-ci plusieurs stratégies d'utilisation sont possibles :

  . soit on spécialise chaque registre pour qu'il ne contienne qu'un seul type de valeur : entier, booléen, etc... (solution adoptée en Algol W ou Algol 68_R).
  . soit on banalise tous les registres (solution que nous avons retenue). Leur gestion pendant la génération est plus compliquée mais elle permet plus de souplesse pour la compilation des expressions et plus d'efficacité à l'exécution.

2) - La gestion des registres consiste en leur allocation et libération. Nous décrivons dans la suite la stratégie d'allocation que nous avons utilisée.

- La libération est consécutive à celle des chaînes de descripteurs dont ils forment l'élément terminal. Cette libération a lieu lorsque la valeur décrite par la chaîne n'est plus nécessaire à la suite de l'élaboration.

- L'allocation peut nécessiter de produire du code pour sauvegarder le contenu d'un ou plusieurs registres lorsqu'il n'est pas possible de satisfaire une demande avec les seuls registres disponibles à cet instant là. Pendant la génération de code, des descripteurs d'un type particulier (apparaissant toujours en fin de chaîne) les décrivent. Chacun contient un accès au descripteur décrivant la valeur contenue dans le registre. On peut ainsi, lors d'une sauvegarde de registre, produire le code nécessaire au rangement du contenu du registre dans la zone d'évaluation. Lors d'une telle opération il faut aussi mettre à jour la chaîne de descripteurs qui ne doit plus décrire une valeur en registre mais une valeur rangée dans la zone d'évaluation.

4.1.2.2 - Etudions brièvement, l'incidence de la technologie du calculateur IBM 360 sur la gestion des registres et les solutions retenues pour notre compilateur.

Les registres sur ce calculateur sont divisés en deux groupes, les registres généraux, pouvant servir de base (en général), et les registres flottants. Un registre général peut être utilisé pour simuler les indirections, (voir paragraphe 4.1.1).

Une autre contrainte concerne les paires de registres :

les opérations de multiplication et de division utilisant les registres généraux nécessitent l'allocation d'une paire de registres (le registre opérande codé dans les instructions étant soit le registre de numéro pair, soit le registre de numéro impair).

Nous avions le choix entre deux solutions pour la gestion des registres flottants et généraux banalisés :

- soit allouer par paire systématiquement et simplifier cette gestion,

- soit n'allouer par paire que lorsque cela s'avère nécessaire et satisfaire ainsi, avec le même nombre de registres un plus grand nombre de demandes. Nous avons choisi la dernière solution.

Les registres à la compilation sont répartis sur quatre files :

- file des registres généraux banalisés (flottants) disponibles,

- file des registres généraux banalisés (flottants) alloués.

Ces files regroupées suivant la nature des registres sont représentées et gérées suivant une technique classique analogue à celles que présente D.E. Knuth [KNUTH-1]. Indiquons maintenant la stratégie que nous avons utilisée pour l'allocation des registres.

a) - Pour allouer un registre simple, on explore la file des registres disponibles Si la file est vide, on alloue, après sauvegarde de son contenu (dans la zone d'évaluation), le registre le plus anciennement occupé.

b) - Pour allouer une paire de registres, on explore aussi la file des registres disponibles. Plusieurs cas sont à considérer :

- il existe une paire de registres libres,
- il n'existe plus de paires mais il y a encore des registres libres. On sauvegarde un registre occupé, de manière à obtenir une paire avec un des registres disponibles. Un algorithme affiné peut même choisir un couple (registre disponible - registre associé occupé) tel que le registre associé soit le plus ancien des registres occupés parmi tous les couples possibles.
- il n'existe aucun registre disponible. On choisit la paire la plus anciennement occupée et on produit le code réalisant la sauvegarde des valeurs s'y trouvant.

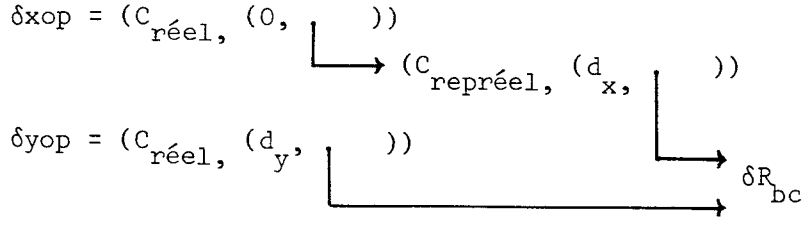
Une telle stratégie peut paraître bien sophistiquée d'autant plus qu'elle suppose un ordre initial de rangement des registres dans les files. D'autre part, pendant la traduction d'un quadruplet, il est préférable de demander l'allocation des paires de registres avant celle des registres simples afin de minimiser les sauvegardes. Grâce à la structure du processus de gestion des registres, nous pouvons aisément changer de stratégie. Ceci nous permet d'effectuer des études expérimentales qui sont les seules à pouvoir confirmer ou infirmer le choix que nous avons fait (le gain d'efficacité devant compenser la complexité de l'algorithme).

4.1.2.3 - Exemple :

Soit le programme Algol 68 suivant :

```
début
    réel y, z, rep réel x,
    ent a, b, c ;
    ...
    z := x + y ;
    c := a * b
fin
```

1) - Considérons l'addition $x + y$. Le quadruplet associé pendant la génération est : (+, δxop, δyop,) où δxop et δyop sont les descripteurs des valeurs opérandes de l'addition. Si δR_{bc} est le descripteur décrivant le registre R_{bc} contenant la base de l'environnement courant (registre permanent), nous avons :



La séquence de code produite est :

L	R,	$d_x (R_{bc})$	R est un registre général demandé par le processus de codage pour produire les indirections.
LD	F,	O(R)	F est un registre flottant demandé par le processus de traduction. Après cette instruction R est libéré.
AD	F,	$d_y (R_{bc})$	F contient le résultat de l'opération.

Le descripteur de la valeur résultat est :

$\delta = (C_{réel}, (Dreg, \ldots))$ → descripteur du registre flottant F

Dreg indique qu'il s'agit d'une valeur contenue dans un registre.

2) - Considérons la multiplication $a * b$. Le quadruplet associé pendant la génération est : (*, δaop, δbop,). Nous avons :

$\delta aop = (C_{ent}, (d_a, \ldots))$ → δR_{bc}

$\delta bop = (C_{ent}, (d_b, \ldots))$ → δR_{bc}

La séquence de code produite est :

L R_{2p+1}, $d_a(R_{bc})$ la paire de registres généraux R_{2p}, R_{2p+1} a été allouée au processus de traduction.

M R_{2p}, $d_b(R_{bc})$ après cette instruction le registre R_{2p} est libéré.

Le descripteur de la valeur résultat est :

$\delta = (C_{ent}, (Dreg, \ldots))$ → descripteur du registre général R_{2p+1}

4.1.3 - Traduction des quadruplets -

La traduction des quadruplets utilise les processus de gestion des registres d'une part, pour disposer de registres temporaires (par exemple pour le quadruplet + :=) ou de registres contenant une valeur significative pour le contexte d'évaluation (exemple quadruplet de code +). Elle utilise d'autre part, le processus de codage des instructions.

On peut distinguer dans la traduction deux types de quadruplets :

- ceux utilisés dans les constructions générales,
- ceux, utilisés dans les opérateurs standard.

Pour ces derniers, la distinction entre deux opérateurs notés fonctionnellement par le même quadruplet n'est effectuée que par l'intermédiaire du mode contenu dans le descripteur de ses opérandes. La conception modulaire de la génération de code, permet en modifiant uniquement ce processus de traduction, de changer des séquences de code associées à un quadruplet ou d'insérer de nouveaux qua-

druplets (au niveau des opérateurs d'un prélude par exemple). Ceci n'est pas possible, en général, sur les compilateurs usuels (PASCAL, ALGOL W [BAUER-AW] par exemple) pour lesquels la génération d'instructions-machine est disséminée dans tous les traitements sémantiques du compilateur.

Dans notre compilateur, le processus de traduction des quadruplets repose sur un principe un peu différent [68SEMI]. En effet nous avons défini un langage de traduction formé d'"instructions de codage". Chaque quadruplet est transformé automatiquement en une séquence de telles instructions dont l'interprétation fournit le code objet. Ce langage permet d'étendre le jeu d'instructions et de données du calculateur réel pour obtenir une machine (abstraite) d'un niveau supérieur. Nous avons abandonné pour ceci la conception descendante pour une conception ascendante. Le processus de traduction des quadruplets se ramène à un processus d'interprétation des instructions de codage. L'objectif initial n'était pas de reprendre l'approche du problème de génération de code faite dans UNCOL [STRO-WEG] (souvent jugée peu réaliste) où un langage intermédiaire universel était introduit, mais d'utiliser une même représentation pour plusieurs langages intermédiaires. Nous pensions ainsi apporter quelques éléments dans le sens d'une automatisation de la production de code.

Un principe analogue est proposé par ERSHOV [ERSHOV] pour construire simultanément plusieurs compilateurs (des langages PL/1, SIMULA67, ALGOL68), ou par WILCOX pour le compilateur du langage PLC [WILCOX]. Dans les deux cas ce langage intermédiaire sert de base à une phase d'optimisation.

Le langage de codage que nous utilisons est formé d'instructions de trois types :

- les instructions de gestion des ressources. Il s'agit essentiellement de la gestion des registres et de celle de la zone statique.
- les instructions permettant de produire des opérations sur les valeurs simples.

Exemples : - addition entre valeurs entières, réelles, etc...
- utilisation des étiquettes (qui peuvent être considérées comme des valeurs simples pendant la génération de code).

- les instructions de contrôle nécessaires à ce langage.

Bien que l'idée d'un tel langage pour représenter les quadruplets reste séduisante sa mise en oeuvre dans le cadre de notre projet qui ne comporte l'écriture que d'un seul compilateur n'a pas été très satisfaisante. Le nombre des instructions de codage est élevé (une cinquantaine) ; la taille de l'ensemble du traducteur est importante et provient essentiellement de l'interprète de ces instructions de codage. La place occupée par les séquences d'instructions à interpréter est également importante bien que partiellement due au nombre élevé d'opérateurs définis dans le prélude standard. Enfin le temps d'interprétation augmente inutilement le temps de compilation.

Dans le cas de notre projet, nous pouvons gagner de la place au niveau du compilateur et du temps au niveau de la compilation, en renonçant au principe du langage intermédiaire et en supprimant l'étape d'interprétation des instructions de codage. Nous ne négligeons pas pour autant, les aspects intéressants de portabilité et de "modularité" du générateur de code ; nous ne perdons même pas, les avantages offerts par un langage intermédiaire pour effectuer des optimisations globales sur le programme, puisque nous disposons d'un texte intermédiaire formé par l'arbre abstrait décoré.

4.2 - Le chargeur -

Nous avons utilisé ce passage supplémentaire (qui peut être évité si l'on engendre directement le code sous une forme non translatable en mémoire), pour effectuer des tâches optionnelles. Le chargeur est conçu spécialement pour le compilateur. Nous espérons qu'il est, d'une part plus efficace que le chargeur fourni par le système opératoire, et d'une utilisation plus agréable pour l'utilisateur ; ce qui à notre avis compense la non-compatibilité du programme objet produit, avec les autres modules objets existant dans le système.

L'existence de ce chargeur spécialisé permet, nous l'avons vu, d'alléger le processus de production de code, de la gestion des étiquèttes et d'un adressage propre au calculateur.

Les tâches optionnelles du chargeur sont de deux types :

- une partie du code peut n'être chargé que sur option, bien qu'elle soit toujours produite. Le code de vérification des débordements d'indices de tableaux, par exemple, est traité ainsi. On peut envisager également de ne charger le code de copie des tableaux, dans les appels de routine, que sur option. Dans un

contexte d'aide à la mise au point de gros programmes on peut ainsi éviter de nombreuses compilations.

- des appels à des routines de trace (appartenant à la bibliothèque) peuvent être insérés dans le code chargé, et servir de base à un système plus complet d'aide à la mise au point. Un tel système présente un grand intérêt pour notre compilateur qui fonctionne dans un environnement conversationnel.

Remarque :
Des numéros d'unité introduits dans le texte source peuvent servir à "repérer" le chemin suivi lors d'une évaluation. Sur option, un numéro "d'unité courante" est mis à jour dynamiquement au cours de l'élaboration du programme.

La forme des instructions produites par le générateur ayant été décrite précédemment, précisons la forme du programme objet à partir duquel s'effectue le chargement. Le programme objet est sauvegardé sur un fichier à accès direct, (les séquences de code n'étant pas obligatoirement chargées dans l'ordre où elles sont produites) constitué d'enregistrements de taille fixe. Le premier enregistrement est un répertoire décrivant les autres enregistrements constituant le programme. Ces enregistrements contiennent la table des constantes, la table des formats, la table de références pour le ramasse-miettes, le code du programme principal, et enfin celui de chacune des routines. Le répertoire contient également des informations identifiant le programme compilé.

Les fonctions d'édition de liens sont particulières à cause de la nature même du langage Algol 68 et de l'infinité de modes possibles. L'édition de liens de plusieurs modules compilés séparément, à partir de programmes source écrits dans deux langages distincts (Algol 68 et un autre) ne pose pas de problèmes nouveaux, seule la non-compatibilité des modules objets produits peut compliquer cette édition.

Par contre le chargement de plusieurs modules "Algol 68" est plus complexe et n'est plus un problème d'édition de liens, mais de compilation séparée. Si l'on ne veut pas utiliser une représentation à l'exécution de la table des modes, ce qui nécessiterait de reconsidérer une partie de la génération de code, on peut envisager des solutions comme celle proposée par I. Currie qui permet de compiler un programme dans un certain "contexte" (*keep*) [CUR-68R].

L'exemple complet donné en annexe permet de préciser, entre autres, la forme du code objet produit, la gestion des registres, les transformations effectuées par le chargeur. Les spécifications complètes de ces traducteurs sont fournies par les notices techniques.

III.7 - EVALUATION DU COMPILATEUR -

Nous donnons en conclusion quelques résultats quantitatifs sur le compilateur qui nous a permis de concrétiser la méthode que nous venons de décrire.

Pour être significative l'évaluation porte sur la version opérationnelle du compilateur pour le calculateur IBM 360, actuellement en cours de réécriture, et non sur la version prototype. A titre indicatif signalons que l'utilisation du langage PL360, pour l'écriture de l'ensemble du compilateur (la version prototype est écrite en PL360 et PL/1), associée à l'introduction du préchargement des tables avec les modes, opérateurs et objets du prélude standard, a permis de diviser par six le temps de compilation, sans tenir compte des autres modifications apportées lors de la réécriture.

Les résultats quantitatifs fournis, sont difficiles à interpréter en dehors de tout contexte, la technologie des calculateurs, les langages compilés, les systèmes opératoires avec lesquels fonctionnent les compilateurs, les services mêmes, attendus de ceux-ci étant toujours très différents. C'est pourquoi nous précisons avec soin, dans ce chapitre le contexte de chaque évaluation.

1 - CARACTERISTIQUES DU COMPILATEUR -

Le compilateur est supporté par le système de temps partagé CP-CMS. Ce choix a été essentiellement motivé par la possibilité d'utiliser l'environnement conversationnel fourni par ce système. L'architecture du compilateur étant indépendante d'un système particulier, il est aisément transportable dans le système OS-MVT avec lequel fonctionne également le calculateur IBM 360.

Le compilateur n'a été écrit que pour des objectifs de recherche. Nous n'avons donc pas sacrifié le langage pour obtenir un compilateur plus performant. Avec ce compilateur, nous n'avons pas, non plus, voulu favoriser certaines classes d'utilisateurs (étudiants, par exemple) ou d'applications. Notre but était double :

a) - étudier les problèmes posés par l'écriture d'un compilateur ; de ce fait, il a été conçu de manière très modulaire, pour pouvoir facilement changer certains algorithmes (au niveau de la production de code par exemple) ou évaluer certaines techniques utilisées. Ce dernier travail est à peine amorcé. Il est également possible d'introduire une phase d'optimisation globale.

b) - étudier le langage Algol 68 ; nous n'avons imposé aucune restriction au langage. La version actuelle du compilateur n'est toutefois pas encore complète entre autres au niveau des fonctions standard . L'architecture que nous avons présentée, nous permet de compléter le compilateur au fur et à mesure de la mise au point des modules, sans perturber ce qui existe. Par exemple les tableaux flexibles ne sont pas implantés au niveau de la génération de code bien que les copies dynamiques de valeurs soient déjà implantées et utilisées pour les déclarations d'identité. De la même manière, tous les opérateurs standard ne sont pas encore introduits dans la génération de code.

Notons enfin qu'au niveau du langage de représentation, le "bilinguisme" français anglais est réalisé, et peut même être étendu simplement. Le langage de représentation "anglais" est celui qui est défini par le Rapport ; le langage de représentation français est commun avec celui qui est utilisé tant à Rennes qu'à Grenoble dans le cadre du projet SFER. (Il est analogue à celui que préconise le groupe de travail Algol 68 de l'AFCET).

Nous donnons maintenant quelques évaluations numériques sur la version actuelle du compilateur.

- Encombrement mémoire du compilateur -

Il est écrit essentiellement dans le langage PL360. La taille du code associé aux différents passages et phases est la suivante, l'unité de mesure étant l'octet :

Code du noyau : 35 K

Code associé au premier passage : 40 K

Code associé au deuxième passage : 40 K

Code associé à la phase de décorations (identification, modification) : 25 K

Code associé à la phase de génération : 60 K

La taille du code du chargeur est de 16 K octets. La taille minimale de la bibliothèque, indispensable à l'exécution d'un programme est de 12 K octets. Dans cette évaluation on ne tient pas compte des fonctions de bibliothèque du prélude standard. La taille des routines de trace nécessaires à la mise au point du compilateur, est comprise dans les évaluations précédentes.

Il n'est pas utile que ce code soit simultanément présent en mémoire ; on peut envisager un chargement dynamique pour chaque passage, ce qui ramène à environ 120 K octets l'espace mémoire nécessaire pour le code du compilateur. Dans cette évaluation, il n'est pas tenu compte de l'encombrement des tables et des textes intermédiaires qui dépendent du programme à compiler (un encombrement moyen serait de 100 K). Le temps de compilation est alors grévé du temps de chargement dynamique.

- Espace mémoire nécessaire à la compilation d'un programme -

Il est occupé par les textes intermédiaires et les tables. Nous donnons, dans la suite, quelques éléments d'évaluation à partir d'un exemple. Rappelons seulement que l'on a deux textes intermédiaires :

- le texte postfixé dont la forme segmentée se prête à une sauvegarde sur support externe.
- une représentation de l'arbre abstrait construite par segments, pour réduire l'encombrement mémoire.

Enfin, pour disposer de l'ensemble du prélude standard, 30 K octets au minimum, sont nécessaires pour les tables.

- Rapidité de compilation -

Il est difficile de donner une évaluation qui ait un sens. Le comptage du nombre d'unités ("instructions de base" du langage Algol 68) de programme compilées par unité de temps nous a paru être la meilleure solution dans notre cas. Le comptage du nombre de cartes, généralement utilisé dans d'autres compilateurs, ne réflète guère la réalité en Algol 68, la "densité sémantique" pouvant énormément varier d'une carte à l'autre. Le temps de compilation est de l'ordre de cent unités à la seconde.

2 - ELEMENTS DE COMPARAISON AVEC D'AUTRES COMPILATEURS -

Le but de ce paragraphe, est de situer le compilateur que nous avons construit par la méthode présentée précédemment, par rapport à d'autres compilateurs utilisant d'autres techniques, et compilant d'autres langages. La comparaison n'est faite ici qu'à partir d'un seul exemple que nous avons voulu significatif, les résultats obtenus ont toutefois pu être confirmés sur d'autres cas.

Cette comparaison est délicate car nous ne voulons évaluer que le compilateur et non le langage. Signalons que des comparaisons partielles entre les compilateurs des principaux langages évolués ont été faites dans **[WICHMA-2]**[WICHMA-3]. L'étude comparative porte sur la compilation et l'exécution d'un exemple pris dans le rapport de définition du langage Algol 68.

Nous précisons les conditions sous lesquelles se sont effectuées les comparaisons.

- Nature de l'exemple -

L'exemple a été choisi pour qu'il n'utilise qu'un sous-ensemble commun à tous les langages et non des concepts particuliers à l'un d'entre eux. Nous ne voulons pas comparer les langages mais les compilateurs. Ce sous-ensemble commun est réputé être raisonnablement efficace dans tous les langages. Les programmes servant à la comparaison n'utilisent que peu d'appels aux bibliothèques de fonctions standard . L'exemple choisi permet de calculer la somme d'une série par la méthode d'Euler. Cet algorithme est exprimé sous la forme que nous avons estimée être la plus adaptée au langage.

- Compilateurs et calculateurs -

Nous avons choisi trois compilateurs pour un même calculateur, IBM 360, de manière à éliminer ses particularités technologiques. Les trois compilateurs sont supportés par le même système CP-CMS, pour que son influence intervienne de manière analogue. Nous n'avons pas pu éliminer dans notre cas l'écart dû aux différences de charge du système et qui peut atteindre jusqu'à dix pour cent des temps mesurés. Les conséquences de cette erreur sont toutefois limitées puisque nous ne voulons montrer qu'une "vraisemblance" et non établir un classement.

Les langages compilés sont tous algorithmiques et de haut niveau : Algol 68, Algol W, PL/1. Dans les trois cas aucune optimisation globale sur le programme n'a été effectuée.

On peut remarquer que les trois compilateurs choisis ne rendent pas tous les mêmes services et ne sont pas tous conçus avec les mêmes objectifs. Le compilateur Algol W est plutôt orienté vers l'enseignement et les applications moyennes. Les objectifs du compilateur PL/1 sont plus ambitieux. Leur architecture et les techniques de compilation utilisées sont très différentes. Par exemple, le compilateur Algol W utilise un seul texte intermédiaire, et une méthode d'analyse ascendante fondée sur les techniques de précédences.

- Eléments de comparaison -

La comparaison ne porte pas uniquement sur le temps de compilation mais également sur la taille du code produit et le temps d'exécution de ce code. Les problèmes de détection d'erreurs, très liés à la méthode d'analyse syntaxique utilisée, ou d'outils d'aide à la mise au point n'entrent pas dans la comparaison. Le lecteur trouve en annexe (annexe 2) les programmes utilisés ainsi que des tableaux de comparaison abondamment commentés.

Indiquons l'exemple ayant servi à ces comparaisons.

Cet exemple est tiré du Rapport (chapitre 11, §11.5).

```
début
    proc euler = (proc (ent) réel f, réel eps, ent fois) réel :
        début
            ent n := 1, t ; réel mn, ds := eps ; [1 : 16] réel m ;
            réel somme := (m [1] := f(1)/2) ;
            pour i depuis 2
                tantque
                    (
                     t := si abs ds < eps
                             alors
                                 t+1
                             sinon
                                 t
                         fsi.
                    ) ≤ fois
                faire
                    mn := f(i) ;
                    pour k jusqua n
                        faire
                            mn := ((ds := mn) + m[k]/2) ;
                            m[k] := ds
                        fait ;
                    somme + := début
                                   ds :=
                                       si abs mn < abs m[n] & n < 16
                                          alors
                                              n + := 1 ;
                                              m[n] := mn ; mn/2
                                          sinon
                                              mn
                                          fsi
                               fin
                fait
                somme
        fin co fin de la procédure euler co ;
```

```
imprimer (euler (
                (ent i) réel : si odd i alors -1/i sinon 1/i fsi,
                1.0 e-5,
                2
               )
         )
fin
```

Nous pouvons remarquer, à partir des évaluations données dans l'annexe, que les performances du compilateur obtenu sont (pour cet exemple) comparables à celles des autres compilateurs et restent parfaitement raisonnables.

3 - CONCLUSION -

L'évaluation précédente montre que le compilateur obtenu par la méthode que nous développons ici est acceptable. Pour évaluer cette méthode nous pourrions prendre en compte d'autres éléments, tels que les avantages apportés par une conception descendante. Notre expérience partielle de réécriture, nous permet d'en vérifier l'intérêt ; la documentation du compilateur s'en trouve également simplifiée.

La description, plus ou moins proche de la définition formelle du langage, et les différentes étapes du raffinement à partir de cette description, fournissent également une aide précieuse, quoique partielle dans la mise au point du compilateur.

Il serait enfin, intéressant de réécrire le compilateur en Algol 68 par exemple, pour montrer les possibilités de portabilité ; pour passer d'un calculateur à un autre, il suffirait de modifier le générateur de code et les routines de bibliothèque.

CONCLUSION

La méthodologie que nous présentons dans cette thèse est issue de notre expérience du langage Algol 68, et en particulier de la réalisation d'un compilateur pour ce langage. Il est difficile, au terme de ce travail, de conclure en quelques pages. En effet, les domaines que nous avons abordés sont nombreux : construction de programmes, techniques de validation, conception d'outils d'aide à l'écriture de compilateurs, mais aussi expérience d'un travail en groupe. Chacun de ces points mérite une étude approfondie, que nous ne faisons pas ici. Nous nous limitons à conclure sur l'objet même de cette thèse.

Nous espérons apporter des éléments de réponse à la question "comment écrire un compilateur". A cette question, il est souvent répondu par une simple énumération d'outils, souvent relatifs, d'ailleurs, à l'analyse syntaxique hors-contexte, au détriment de la syntaxe contextuelle et de la génération de code. Nous avons montré comment, à partir d'un schéma simple de traduction, fondé sur la mise en oeuvre d'une structure abstraite des programmes, il est possible d'obtenir assez rapidement un compilateur, d'une manière systématique, pourvu que le langage soit "bien défini".

Il importe d'abord de sérier les problèmes. Nous ne saurions trop souligner l'importance de l'expérience, à ce niveau. Il s'agit moins d'avoir une connaissance d'un grand nombre de techniques, que d'avoir le recul nécessaire pour juger de l'importance des problèmes qui se posent. En l'absence de ce recul, beaucoup d'énergie risque d'être dépensée pour résoudre des problèmes secondaires. De plus, il est alors difficile de choisir les solutions les mieux adaptées aux objectifs fixés. La venue à Grenoble de Ian Currie en 1973, a été pour nous l'occasion d'une prise de conscience en ce domaine.

Notre démarche n'a pas été exempte d'erreurs. Comme bon nombre d'écrivains de compilateurs, nous avons commencé par l'étude de l'analyse syntaxique. Il est vrai qu'en Algol 68 il existe là un réel problème, mais qui doit être relégué à sa juste place. Il faut d'abord étudier la sémantique du langage, avec pour objectifs, non seulement la conception d'un système à l'exécution, mais aussi celle d'une structure abstraite pour les programmes. Nous considérons cette structure abstraite comme la charnière du compilateur.

Le premier niveau de réflexion correspond à ce que nous appelons la description de la fonction de traduction. En pratique, une telle description peut être moins détaillée que celle présentée dans cette thèse, qui visait un but pédagogique. Ni l'analyse syntaxique, ni la production d'instructions pour un calculateur particulier, ne doivent faire partie de cette étude. Il s'agit ici d'envisager uniquement, la construction d'une structure abstraite pour le programme, et la génération de code à partir de cette structure. Il est possible pour cela d'utiliser plusieurs formalismes. Dans notre cas, la méthode des attributs,et un langage de notre conception, se sont montrés bien adaptés. Bien que cette description ait été faite une fois écrite la version prototype du compilateur, elle nous a permis de découvrir des erreurs, ainsi que de trouver des solutions plus systématiques à certains problèmes. De plus, elle permet d'avoir une vision d'ensemble du travail, et constitue le point de départ de l'étape suivante, qui est le choix de méthodes particulières, et l'écriture du compilateur.

La description ne préjuge pas de solutions particulières au niveau de la réalisation du compilateur. Par exemple, elle n'exclut pas un compilateur en un seul passage. Dans ce cas, la structure abstraite du programme n'a pas de représentation effective en mémoire, mais elle est induite par l'analyse syntaxique du programme.

Il s'agit à ce niveau de décider de l'organisation du compilateur et de la représentation des éventuels textes intermédiaires, d'une part, des méthodes et des outils de programmation, d'autre part. Ceci comprend le choix d'une méthode d'analyse syntaxique et d'une technique de production de code pour un calculateur particulier, puis le choix d'outils spécifiques, comme un générateur d'analyseurs, et d'un ou plusieurs langages de programmation.

On peut remarquer qu'il est souvent fait, au niveau de l'enseignement de la compilation, une présentation différente des problèmes. L'accent est mis sur l'analyse syntaxique, qui a certes de l'importance, mais qui peut masquer d'autres problèmes, à notre avis plus fondamentaux, comme la mise en oeuvre d'une méthode.

Le travail de conception est plus ou moins aisé, selon la nature du langage à compiler et la qualité de sa définition. Dans le cas du langage Algol 68, la définition a l'avantage d'être précise et complète. Même lorsqu'un choix est laissé à l'écrivain de compilateurs, le point concerné est mis en évidence. Malheureusement, cette définition n'est pas toujours accessible, et la réponse à des questions parfois fondamentales peut nécessiter une longue recherche, qui n'est pas sans risque d'erreurs, comme nous en avons fait l'expérience. L'emploi, pour la définition d'un langage, d'un formalisme directement utilisable dans une implantation, constituerait un progrès appréciable.

Le formalisme des grammaires hors-contexte présente l'avantage d'être connu et facilement utilisable. Par contre, sa puissance d'expression est limitée aux aspects non-contextuels de la syntaxe du langage. Dans une implantation prototype, certains aspects contextuels, exprimés au moyen d'une W-grammaire dans la définition d'Algol 68, peuvent être, par exemple, représentés au moyen d'un système d'attributs sur une grammaire hors-contexte. Dans ces conditions, on peut penser que, si la définition d'un langage est exprimable en termes d'attributs, on a une présomption de la validité du compilateur obtenu en implantant le système d'attributs proposé. Par transformation de ce prototype, il est ensuite possible de produire des compilateurs adaptés à des besoins spécifiques, tout en conservant cette présomption de correction.

Un deuxième intérêt de notre travail est de conduire à la réalisation de compilateurs Algol 68 pour les calculateurs IBM 360 et CII IRIS 80. Cette réalisation fournit les fondements de la méthodologie proposée, et contribue à la connaissance du langage Algol 68. Quelles que soient les critiques que l'on puisse formuler vis à vis de ce langage, il est indéniable que les concepts introduits servent,à l'heure actuelle, de base à toute réflexion sur les langages de programmation. De plus, nous pensons qu'il est simple à utiliser et à enseigner. Cette opinion est étayée par notre propre expérience d'enseignement, tant en milieu universitaire qu'industriel, et celle d'autres personnes, notamment en Angleterre, où le langage Algol 68 connait un certain succès ; ce succès est rendu possible par la réalisation rapide d'un compilateur, disponible sur les calculateurs ICL 1900. Nous souhaitons que les efforts entrepris, tant au niveau de l'enseignement, que de l'écriture de compilateurs, assurent au langage Algol 68 une diffusion à la mesure des services qu'il peut rendre.

REFERENCES

[AFCET-1] Groupe Algol de l'AFCET
Définition du langage algorithmique Algol 68
BUFFET, J. - ARNAL, P. - QUERE, A. - (Editeurs)
Hermann
1972

[AFCET-2] Groupe Algol de l'AFCET
Le langage algorithmique Algol 68
ANDRE, J. - BACCHUS, P. - PAIR, C. - (Editeurs)
Hermann
1975

[AFCET-3] Journée Algol 68
Université de Paris VI
Avril 1973

[BANATRE] BANATRE, J.P.
Structure d'un compilateur Algol 68
Thèse - 1974
Université de Rennes

[BAUER-AW] BAUER, H. - BECKER, S. - and GRAHAM, S.
Algol W implementation
Technical Report N°. CS 98
Stanford University
Mai 1968

[BORDIER] BORDIER, J.
Méthodes pour la mise au point de grammaires LL(1)
Thèse - Janvier 1971
Faculté des Sciences de Grenoble

[BO-SOFT] BOUCKAERT, M. - PIROTTE, A. - and SNELLING, M.
SOFT : A tool for writing software
M.B.L.E. Laboratoire de Recherche - Bruxelles
Report R 212
1973

[BOU-DUBY] BOUSSARD, J.C. - DUBY, J.J. - (Editeurs)
Rapport d'évaluation d'Algol 68
IMAG, Grenoble et Centre Scientifique IBM France
Juillet 1970

[BRANQU-1] BRANQUART, P. - CARDINAEL, J.P. - LEWI, J. - DELESCAILLE, J.P. - and VAN BEGIN, M.
Data structure handling in Algol 68 compilation
International Conference on Algol 68
Informal Implementer's Interchange
Winnipeg - Juin 1974

[BRANQU-2] BRANQUART, P. - CARDINAEL, J.P. - LEWI, J. - DELESCAILLE, J.P. - and VAN BEGIN, M.
An optimized translation process and its application to Algol 68
Part I : General principles - Septembre 1972
Part II : Block constructions - Janvier 1974
Part III : Other constructions - Février 1974
Part IV : Machine code generation - Mai 1974
M.B.L.E. Research Laboratory
Bruxelles

[BRANQU-3] BRANQUART, P. - LEWI, J. - et al.
Reports
R117, R121, R123, R125, R130, R133, ...
Technical Notes
N60, N62, N66, N68, ...
M.B.L.E. Research Laboratory
Bruxelles

[BRE-DEL] BRETAGNOLLE, B. - DELAUNAY, M.
Autocompilation de CDL
Version de l'analyse LL(1) en PL360
Notes internes
1974 - 1975

[BUFFET] BUFFET, J.
Etudes préliminaires à la compilation d'Algol 68
Thèse - 1972
Université des Sciences et Techniques de Lille 1

[CAPO-MU5] CAPON, P.C. - MORRIS, D. - ROHL, J.S. - and WILSON, I.R.
The MU5 compiler target language and autocode
Department of Computer Science
University of Manchester - 1974

[CHAS-COL] de CHASTELLIER, G. - et COLMERAUER, A.
W-grammar
Projet de traduction automatique
Proceedings ACM National Conference
1969 - p. 511-518

[CLARK] CLARK, D. - GRAHAM, P. - SALTZER, J. and SCHROEDER, M.
The classroom information and computing service
MAC TR80 MIT PRESS
1971

[CLE-UZGA] CLEAVELAND, I.C. - and UZGALIS, R.C.
What every programmer should know about grammar
Modeling and measurement note N° 12
Computer Science Department
School of engineering and applied science
University of California, Los Angeles
Mars 1973

[COUSOT] COUSOT, P.
Définition interprétative et implantation de langages de programmation
Thèse - Décembre 1974
Université Scientifique et Médicale de Grenoble

[CREECH] CREECH, B.
Architecture of the B6500
Infotech Report2
1971

[CURRIE] CURRIE, I.F.
Inside the Algol 68-R compilers
International conference on Algol 68 implementation
University of Manitoba
Winnipeg
Juin 1974 - p. 1-11

[CUR-68R] CURRIE, I.F. - BOND, S.G. - and MORISON, J.D.
Algol 68-R
Algol 68 implementation
PECK, J.E.L. (Ed.)
North-Holland
1971 - p. 21 - 34

[DAHL-HOA] DAHL, O.J. - DIJKSTRA, E.W. - and HOARE, C.A.R.
Structured programming
Academic Press, New-York - 1972

[DELAUNAY] DELAUNAY, M.
Version de l'analyse LL(1) utilisable sur IRIS 80
Publication interne
Février 1975

[DIJKSTRA] DIJKSTRA, E.W.
Notes on structured programming
Report NR 241 Technische Hogeschool
Eindhoven - 1969

[ERSHOV] ERSHOV, A.P.
A multilanguage programming system oriented to languages
description and universal optimization algorithms
Algol 68 implementation
PECK, J.E.L. (Ed.)
North-Holland - 1970

[EXEL] EXEL, M.
De l'optimisation des programmes
Thèse - 1975
Université Scientifique et Médicale de Grenoble

[FAN-FOLD] FANG, I.
- FOLDS, a declarative formal language definition system
PHD Thesis - 1972
Computer Science Department
Stanford University
- TR STAN CS-72-329
1972
- Séminaire Structures et Programmation
(p. 275-290)
IRIA Rocquencourt
1973

[FELD-TWS] FELDMAN, J. - and GRIES, D.
Translator Writing Systems
Communication ACM Volume 11, Number 2
Février 1968

[FOSTER] FOSTER, J.M.
A Syntax Improving Device
Computer Journal
Mai 1968

[GRIES] GRIES, D.
Compiler Construction for digital computers
WILEY, J. and sons Inc. (Ed.)
1971

[GRIFF-1] GRIFFITHS, M.
How to use the grammar transformer
Rapport interne
IMAG Grenoble
1968

[GRIFF-2] GRIFFITHS, M.
Analyse déterministe et compilateurs.
Thèse - 1969
Faculté des Sciences de Grenoble

[GRIFF-3] GRIFFITHS, M.
Relationship between definition and implementation of a language
Software Engineering-An advanced Course (Munich 1972)
Lecture notes in Computer Sciences. Edited by G. Goos and J.Hartmanis
Springer Verlag 1975.

[III-74] International Conference on Algol 68 implementation
Department of Computer Science
University of Manitoba, Winnipeg
Juin 1974

[III-75] International Conference on Algol 68
Department of Computing and Information Sciences
Oklahoma State University, Stillwater
Juin 1975

[IRIASFER] Etude et réalisation d'un compilateur Algol 68 sur ordinateur
CII - IRIS 80 sous système SIRIS 8
Contrat SESORI - Projet SFER
N° 74-70

[KNUTH-1] KNUTH, D.E.
The art of computer programming - Fundamental algorithms
Chapitre 2
Addison Wesley Publishing Company

[KNUTH-2] KNUTH, D.E.
Semantics of context free languages
- Math-Systems Theory Vol. 2 N° 2, pp. 127-145
1968
- Math-Systems Theory Vol. 5 N° 1, pp. 95-96
1971

[KNUTH-3] KNUTH, D.E.
Top down syntax analysis
International Summer School on Computer Programming
Copenhague - Août 1967.

[KNUTH-4] KNUTH, D.E.
Examples of formal semantics
Symposium on semantics of algorithmic languages
Lecture notes in Mathematics n° 188
Springer Verlag - 1971

[KOSTER-1] KOSTER, C.H.A.
On infinite modes
Algol Bulletin N° AB 30 - pp. 86-89
Février 1969

[KOSTER-2] KOSTER, C.H.A.
A compiler compiler
Mathematisch Centrum Amsterdam
Report MR127
1971

[LIN-MEUL] LINDSEY, C.H. - and VAN DER MEULEN, S.G.
- Informal introduction to Algol 68
  Mathematisch Centrum Amsterdam
  Août 1969
- Informal introduction to Algol 68
  North-Holland
  1971

[LINDSEY] LINDSEY, C.H.
Making the hardware suit the language
Algol 68 implementation
PECK, J.E.L. (Ed.)
North-Holland 1970

[LORHO] LORHO, B.
De la définition à la traduction des langages de programmation : méthode des attributs sémantiques
Thèse - 1974
Faculté des Sciences de l'Université Paul Sabatier
Toulouse

[LUCAS] LUCAS, P. - LAUER, P. - and STIGLEITNER, H.
Method and notation for the formal definition of programming languages
IBM Laboratory Vienna
TR 25087 - 1968

[MACCARTH] MAC CARTHY, J.
Basis for mathematical theory of computation
Computer programming and formal systems
BRAFFORT and HIRSCHBERG (Ed.)
North-Holland Amsterdam
1963

[MAILLOUX] MAILLOUX, B.J.
On the implementation of Algol 68
Thèse - 1968
Akademisch Proefschrift
Mathematisch Centrum - Amsterdam

[MUNICH] Compiler Construction - An advanced course
Springer-Verlag - Munich
1974

[ORGA-CLE] ORGANICK, E. - and CLEARY, J.
A data structure model of the B 6700 computer system
ACM Sigplan Notices
1971

[PAIR] PAIR, C.
Définition et étude des bilangages réguliers
Information and Control
1968 - 13, pp. 565 - 593

[PECK] PECK, J.E.L.
Draft of an Algol 68 companion
Department of Computer Science
University of British Columbia
Vancouver
Mars 1971

[RAN-RUSS] RANDELL, B. - and RUSSEL, L.J.
Algol 60 implementation
Academic Press, APIC Studies in Data Processing N° 5
1964

[REZIG] REZIG, R.
Application de la méthode de précédence totale à l'analyse d'Algol 68
Rapport de DEA - 1972
Université Scientifique et Médicale de Grenoble

[SEHTI] SEHTI, R.
Complete register allocation problems
ACM Symposium on Theory of Computing
1973

[SINTZOFF] SINTZOFF, M.
Vérification d'assertions pour des fonctions utilisables comme valeurs et affectant des variables extérieures
Colloques IRIA
Construction, amélioration et vérification de programmes
Arc et Senans
Juillet 1975 - pp. 11 - 27

[STRO-WEG] STRONG, J. - WEGSTEIN, J. - TRITTER, A. - OLSZTYN, J. - MOCK, O. - and STEEL, T.
The problem of programming communication with changing machines
Communications ACM 1 N° 8, pp. 12-18, N° 9 , pp. 9 - 15
1958

[UZGALIS] UZGALIS, P.
Draft : Language changes in revised Algol 68 report
Juillet 1973

[VANWIJ-1] VAN WIJNGAARDEN, A. (Ed.) - MAILLOUX, B.J. - PECK, J.E.L. - and KOSTER, C.H.A.
Report on the algorithmic language Algol 68
Mathematisch Centrum Report MR101 Amsterdam
Octobre 1969

[VANWIJ-2] VAN WIJNGAARDEN, A. - MAILLOUX, B.J. - PECK, J.E.L. - KOSTER, C.H.A. - SINTZOFF, M. - LINDSEY, C.H. - MEERTENS, L.G.L.T. - and FISKER, R.G.
Revised Report on the Algorithmic Language Algol 68
Supplement to Algol Bulletin 36
Technical Report TR74-3
Mars 1974

[VANWIJ-3] VAN WIJNGAARDEN, A.
Orthogonal design and description of a Formal Language
Mathematisch Centrum Report MR76, Amsterdam
1965

[WICHMA-1] WICHMANN, B.A.
Algol 60 Compilation and assessment
Academic Press - London
1973

[WICHMA-2] WICHMANN, B.A.
Ackermann's Function
Working Paper
IFIP WG 2.4.
Berlin
Décembre 1974

[WICHMA-3] WICHMANN, B.A.
Estimating the execution speed of an Algol program
SIGPLAN notices
Août 1972

[VERJUS] VERJUS, J.P.
Nature et composition des objets manipulés dans un système
Thèse - 1973
Université de Rennes

[WILCOX] WILCOX, T.R.
Generating machine code for high level programming languages
Thèse - 1971
Computer Science Department
Cornell University
Ithaca, New-York

[WILLIS] WILLIS, B.
Définition et implantation de la sémantique des langages de programmation
Thèse - 1974
Université Scientifique et Médicale de Grenoble

[WOODWARD] WOODWARD, P.M. - and BOND, S.G.
Algol 68-R Users guide
Division of Computing and Software Research
Royal Radar Establishment
HMSO Londres
1972

[ZOSEL] ZOSEL, M.
A formal grammar for the representation of modes and its application to Algol 68
1971 - Thèse
University of Washington

[68GRE-1] Essai d'implémentation des Entrées-Sorties d'Algol 68
SAYA, H. - et VOIRON J.
Rapport de D.E.A.
Université des Sciences de Grenoble
1970

[68GRE-2] Une grammaire context free d'Algol 68
SIMONET, M.
Congrés d'Informatique AFCET
Paris, 1970 - pp. 5. 3. 119 - 135

[68GRE-3] Architecture d'un compilateur Algol 68 en trois passages
CUNIN, P.Y. - SIMONET, M. - VOIRON, J. - et WILLIS, B.
Congrés d'Informatique AFCET
Grenoble, 1972, Vol.1, pp. 355-368

[68GRE-4] Les modifications Algol 68 - Implantation possible
CUNIN, P.Y. - DELAUNAY, M. - SIMONET, M. - VOIRON, J. et WILLIS, B.
Journée Algol 68
Université de Paris VI
Avril 1973

[68GRE-5] Tree segmentation, coercions and code generation
CUNIN, P.Y. - DELAUNAY, M. - SIMONET, M. - VOIRON, J. - and WILLIS, B.
International Conference on Algol 68 implementation
Informal implementers interchange
Winnipeg
Juin 1974, pp. 81 - 95

[68GRE-6] Coercion detection on an abstract tree
CUNIN, P.Y - DELAUNAY, M. - SIMONET, M. - and VOIRON, J.
Conference : Experience with Algol 68
Liverpool
Avril 1975

[68GRE-7] Code generation from a decorated tree
CUNIN, P.Y. - DELAUNAY, M. - SIMONET, M. - and VOIRON, J.
International Conference on Algol 68
Informal implementers interchange
Stillwater
Juin 1975

[68GRE-8] Une implantation du calcul parallèle en Algol 68
CUNIN, P.Y. - DELAUNAY, M. - SIMONET, M. - et VOIRON, J.
non publié
Septembre 1973

[68SEMI] Séminaires

CUNIN, P.Y. - DELAUNAY, M. - SIMONET, M. et VOIRON, J.

- Construction d'un arbre abstrait décoré dans un compilateur
  IRIA Avril 1975
- Génération de code à partir d'un arbre abstrait décoré
  IRIA Avril 1975
- Arbre abstrait décoré : charnière d'un compilateur
  Nice Avril 1975
- Modèle de génération de code et chargement en Algol 68
  Rennes Mai 1975
- Aspects d'une méthodologie d'écriture de compilateurs
  Grenoble Mai 1975

[68TECH] Notices techniques du compilateur Algol 68 de Grenoble
BRETAGNOLLE, B. - CUNIN, P.Y. - DELAUNAY, M. - SIMONET, M. - et VOIRON, J.

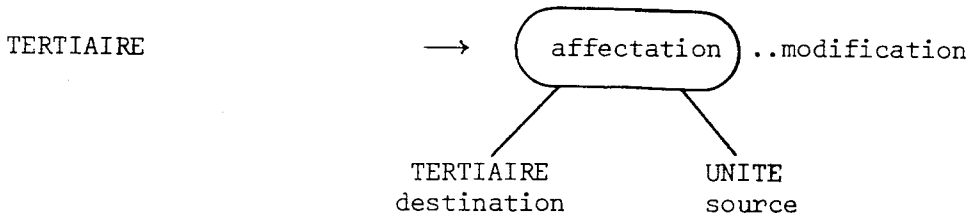
ANNEXE 1

DESCRIPTION DE L'ARBRE ABSTRAIT DECORE -

L'arbre abstrait décoré est décrit par une grammaire d'arborescences. Les décorations utilisées pour la génération de code figurent à droite des noeuds. Les décorations temporaires utilisées lors des processus de construction de l'arbre n'y figurent pas. Elles sont précisées lors de leur utilisation.

Les symboles non-terminaux de la grammaire sont en majuscules, et les noms des noeuds en minuscules. Aux non-terminaux apparaissant en opérande de noeuds, on adjoint un nom de sélecteur, en minuscules, qui sera utilisé pour le traitement de l'arbre abstrait.

Exemple :



Le noeud (affectation) a une décoration modification
et deux opérandes de type TERTIAIRE et UNITE sélectés par les noms destination et source respectivement.

Conventions particulières

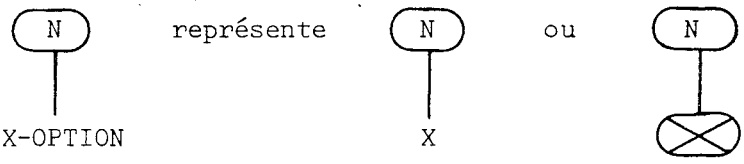
Si X représente un non terminal quelconque,

X-OPTION utilisé en opérande d'un noeud N signifie que cet opérande peut être nil.

X-OPTION → X

→ ⊗

Ce qui signifie que



X-LISTE représente un ensemble ordonné d'arborescences de type X.
Il lui correspond la règle implicite;

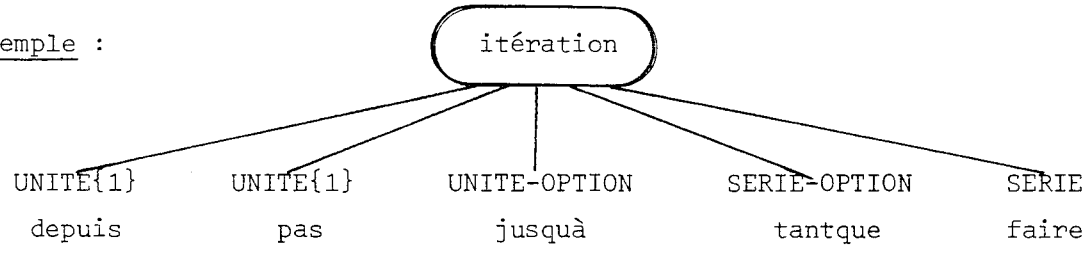
X-LISTE → X

→ X ⊕ X-LISTE

où ⊕ représente la concaténation d'arborescences [PAIR].

X{y y y} signifie que X peut être omis, auquel cas l'arbre correspondant est remplacé par la valeur yyy.

Exemple :



La partie depuis peut être omise. Sa valeur par défaut est 1.
La partie pas peut être omise. Sa valeur par défaut est 1.
La partie jusquà est optionnelle. Sa valeur par défaut est ⊗
La partie tantque est optionnelle. Sa valeur par défaut est ⊗

Grammaire de l'arbre abstrait — Décorations

PROGRAMME → programme particulier

PROPOSITION programme

PROPOSITION → début ..existetiquette

SERIE série

co *début série fin* co

→ collatéral ..nb éléments mode

UNITE-LISTE liste-composant

co *(liste d'unités)* co

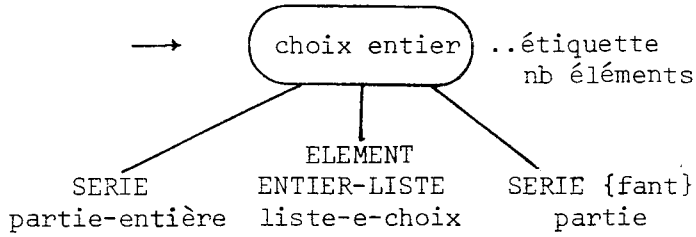
→ choix booléen ..étiquette

SERIE condition — SERIE partie alors — SERIE {fant} partie sinon

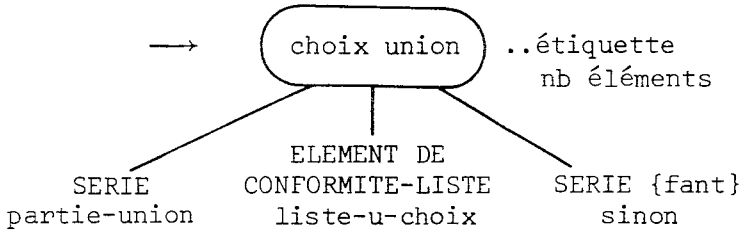
co *si condition alors série sinon série fsi*

Lorsque la partie sinon est omise, la SERIE sinon est remplacée par (fant)

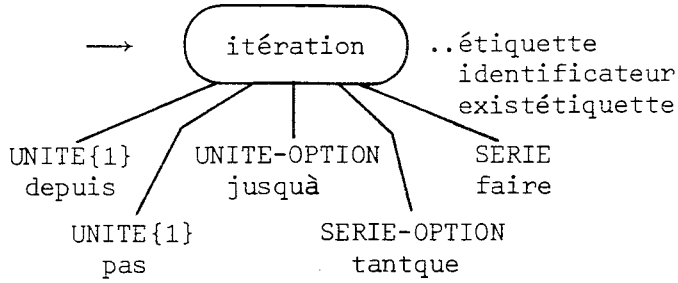
co



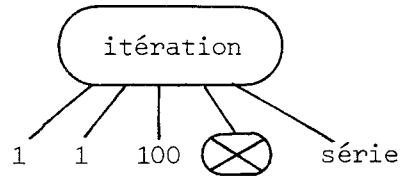
co *cas* e *dans liste d'unités sinon série fcas* co



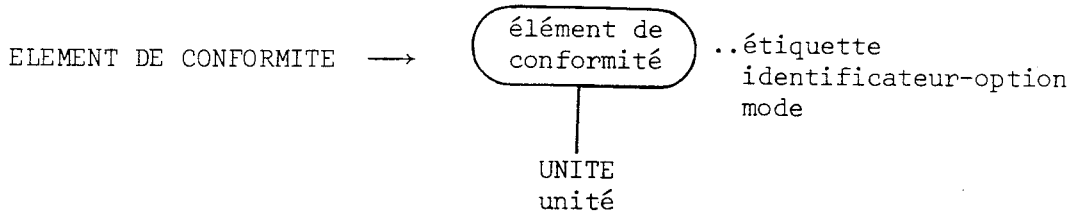
co *cas u dans (réel) : unité 1,*
(ent i): unité 2,
...
sinon série
fcas
co



co *pour i jusquà 100 faire série fait*



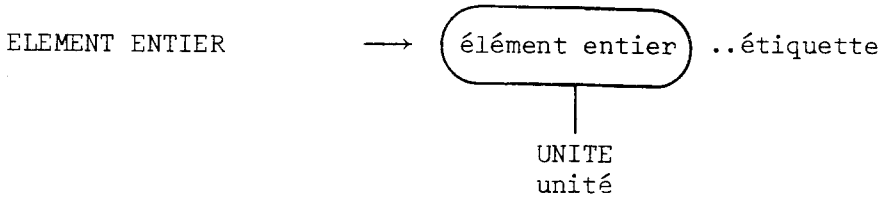
co

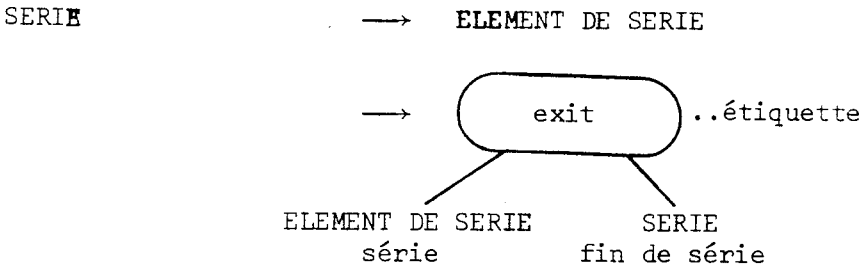


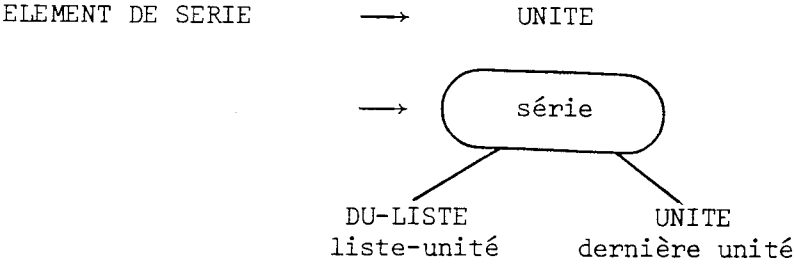
co (*déclareur idf*): *unité*

La décoration *identificateur* correspond à l'identificateur qui peut être omis.

co







co La dernière unité d'une série, ou celle qui précède un acheveur, détermine le mode de la proposition sérielle. Elle est différenciée.

Le noeud (série) n'est pas créé lorsqu'il n'y a qu'une unité.

co

DU ⟶ DECLARATION

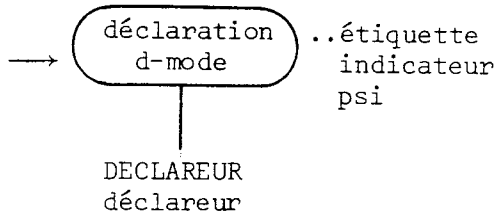
⟶ UNITE ETIQUETABLE

UNITE ETIQUETABLE ⟶ UNITE

⟶ étiquette ..identificateur

UNITE ETIQUETABLE
unité

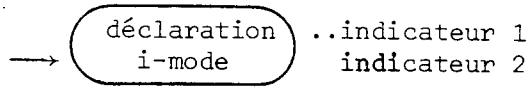
DECLARATION



co *mode indicateur = déclareur*

Le déclareur contient des parties dynamiques (bornes de tableau)

co

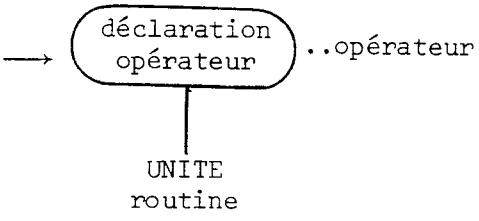


co *mode indicateur 1 = indicateur 2*

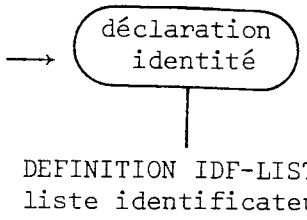
Lorsque le déclareur, associé à *indicateur 2*, a des parties dynamiques.

Les déclarations de mode sans partie dynamique n'apparaissent pas dans l'arbre

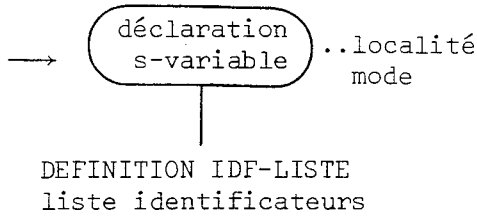
co



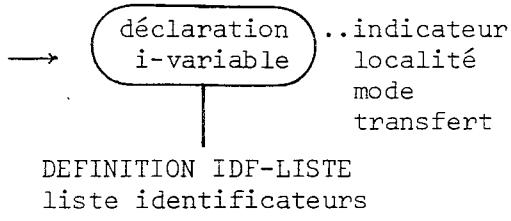
co *op opérateur = routine* co



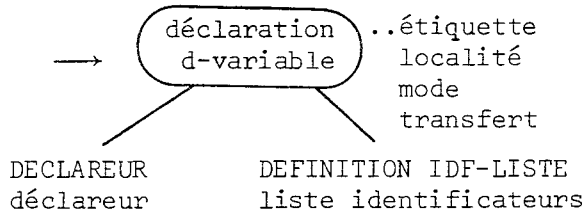
co *réel x = unité, y = unité* co



co *réel x,y ;*
Déclareur sans parties dynamiques
co

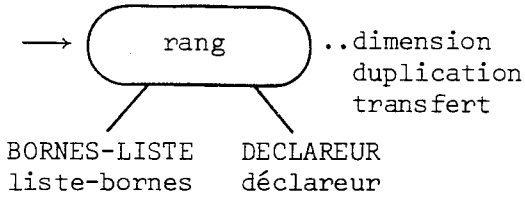


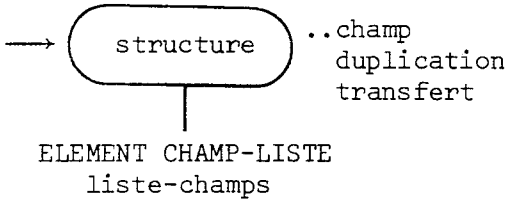
co *m x,y ;*
L'indicateur m contient des parties dynamiques
co Le déclareur, associé à l'indicateur *m*, contient des parties dynamiques

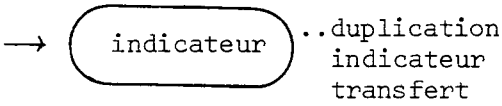


co *[1:n] ent t ;*
Le déclareur contient des parties dynamiques
co

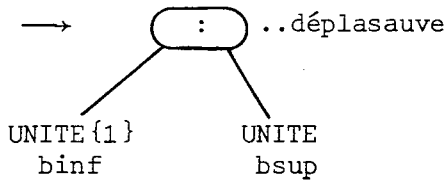
DECLAREUR





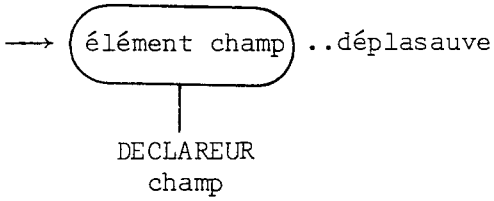


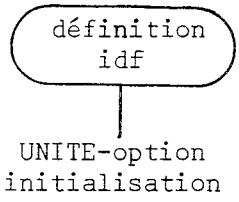
BORNES



co Lorsque la borne supérieure est donnée seule, [5] *ént*, la borne inférieure est 1 par défaut. co

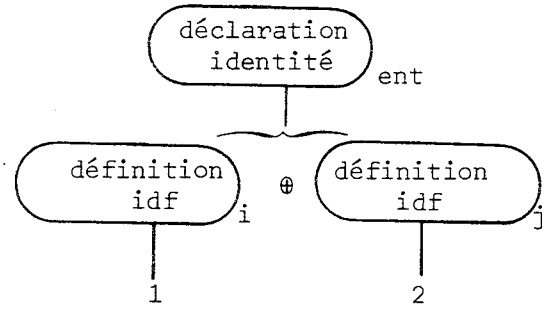
ELEMENT CHAMP



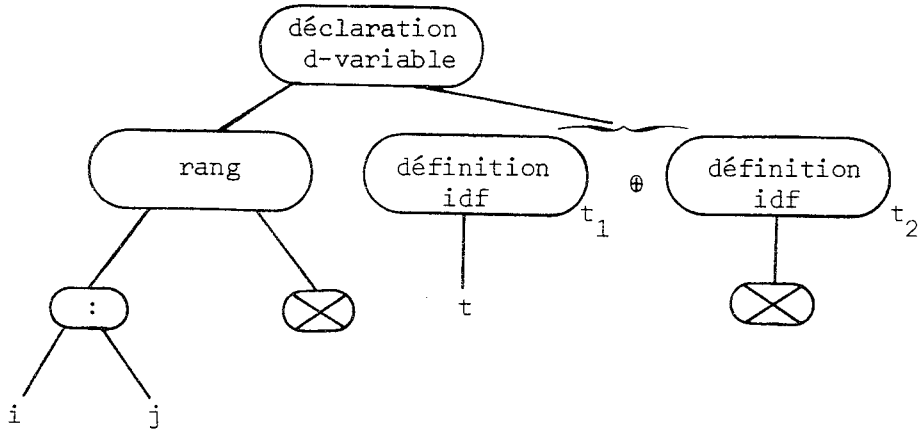
DEFINITION IDF ⟶ 

co Un noeud (définition idf) est créé pour chaque élément d'une liste, dans une déclaration d'identité ou de variable

Exemple : *ent i = 1, j = 2 ;*

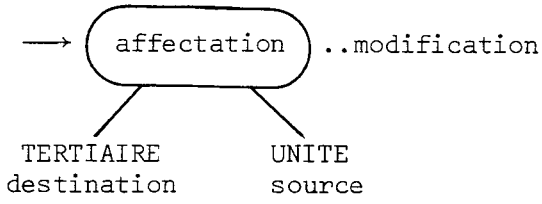


[i:j] réel t1 := t, t2 ;

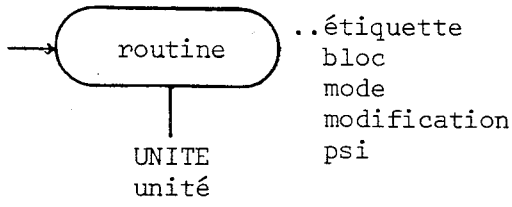


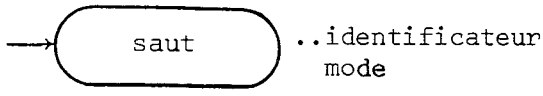
co

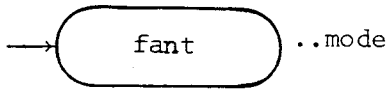
UNITE



co La relation d'identité n'apparaît pas car elle a été transformée en opérateur co

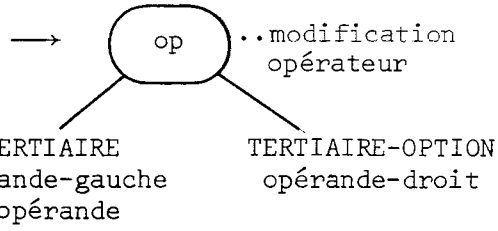






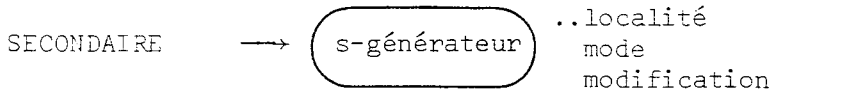
⟶ TERTIAIRE

TERTIAIRE

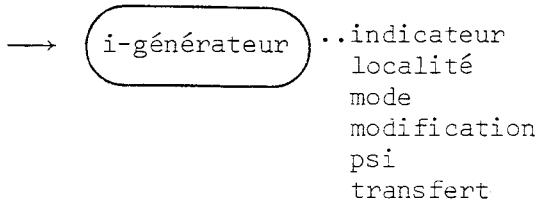


co Pour un opérateur unaire, l'opérande droit vaut ⊗ , et il y a donc un seul opérande co

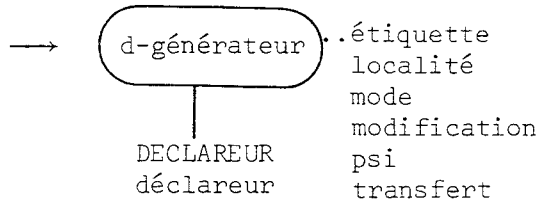
⟶ SECONDAIRE



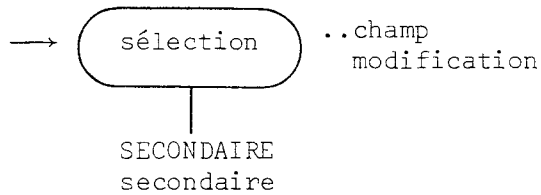
co Déclareur sans parties dynamiques co



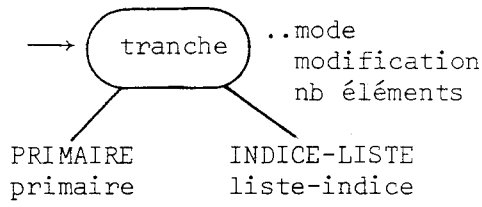
co Déclareur à parties dynamiques co



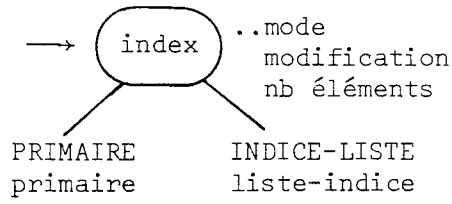
co Déclareur à parties dynamiques co



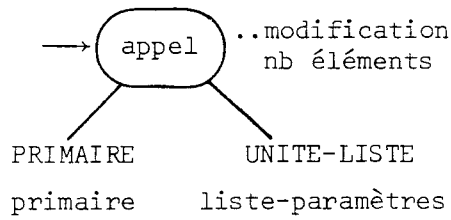
PRIMAIRE



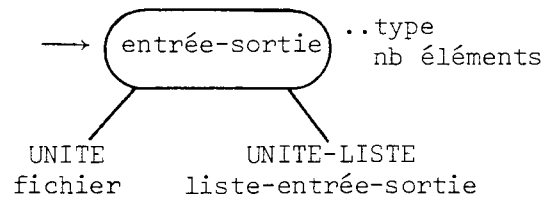
co *t* [*2 : 5*]
t [*3 apd 1*]
co



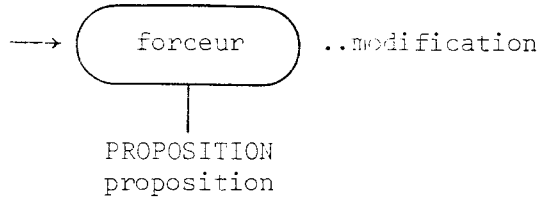
co *t* [*1, j*] co

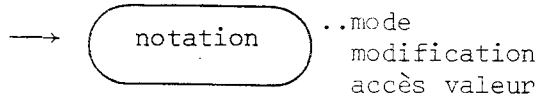


co *p (a, b)* co

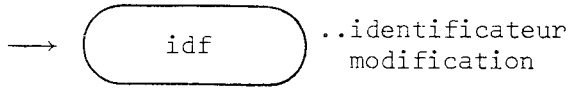


co *inf ((fichier, i, j))* co





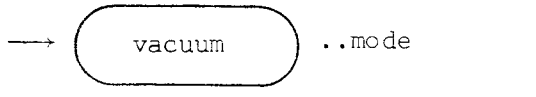
co *1 3*
5.0 3.14
vrai, faux
"a"
co



⟶ PROPOSITION

⟶ SERIE

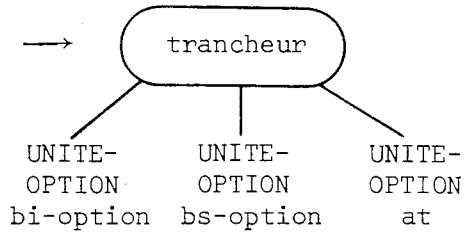
co Lorsqu'une proposition fermée **ne** contient pas de déclarations, le noeud (début) n'est pas créé et on trouve directement la série.
co



INDICE ⟶ Λ

⟶ UNITE

⟶



DECLARATIONS DES DECORATIONS DE L'ARBRE ABSTRAIT -

Les décorations de l'arbre abstrait se répartissent dans les catégories suivantes :

Décorations d'accès et de caractéristiques des valeurs

décoration mode =
    co caractéristiques statiques d'une valeur, c'est-à-dire :
    taille de la partie statique de la représentation,
    facteur d'alignement (si besoin est),
    accès à la représentation, dans le compilateur, du déclareur associé
    ...etc... co

décoration identificateur =
    co décoration associée à l'identificateur possédant la valeur à manipuler.
    Elle correspond à une décoration mode augmentée de toutes les informations concernant l'accès à la partie statique de la représentation de la valeur possédée par l'identificateur (niveau statique, déplacement dans la zone des variables, de l'environnement associé, ...) co

décoration opérateur =
    co décoration associée à l'opérateur possédant la valeur à manipuler.
    Elle est semblable à la décoration identificateur.
    La valeur possédée est de mode procédure et sa représentation (formée de deux pointeurs) est dans la zone des variables de l'environnement contenant la déclaration co

décoration indicateur =
    co décoration associée à l'indicateur, pour lequel est produite une routine évaluant les bornes des tableaux figurant dans le déclareur effectif.
    Elle est semblable à la décoration identificateur.
    S'agissant d'une routine, la représentation correspondante est celle des valeurs de mode procédure co

En plus de ces quatre décorations principales, il existe des décorations remplissant les mêmes fonctions mais d'utilisation plus restreinte.

décoration accèsvaleur =

co sur le noeud (notation) , représente l'accès statique à la représentation de la notation dans la table des constantes co

décoration champ =

co sur les noeuds (structure) et (sélection), décoration associée au champ de la structure.
Elle est analogue à la décoration identificateur, mis à part que l'accès à la représentation de la valeur du champ est calculé statiquement depuis le début de la représentation de la structure co

Décorations pour les déclareurs d'objets à parties dynamiques

Il s'agit de décorations servant au remplissage de la zone de transfert entre la routine évaluant les bornes contenues dans le déclareur de mode et le traitement des objets de ce mode.

décoration transfert =

co décoration indiquant la taille de la zone de transfert correspondant au déclareur du noeud co

décoration duplication =

co décoration reflétant la factorisation (par l'utilisateur) des déclareurs des champs d'une structure. Elle est utilisée dans les noeuds (rang) , (structure) et (indicateur) pour dupliquer une partie de la zone de transfert autant de fois qu'il y a de déclareurs de champs sous-entendus co

décoration déplasauve =

co sur les noeuds (bornes) (éléments de la liste opérande du noeud (rang)) et (élément-champ) (éléments de la liste opérande du noeud (structure)) cette décoration indique le déplacement, dans la zone de transfert, à partir duquel il faut ranger les valeurs de la paire de bornes (cas du noeud (bornes)) ou de l'ensemble des bornes apparaissant dans le déclareur du champ (cas du noeud (élément-champ))

décoration dimension =

co sur le noeud (rang) , cette décoration indique le nombre de dimensions du tableau ce qui permet de connaître la taille de la partie de la zone de transfert que remplit l'évaluation des bornes de ce tableau (indépendamment du déclareur de ses éléments) co

Décoration pour l'allocation

décoration localité =

co pour les déclarations de variables et les générateurs, elle indique si l'allocation est à faire sur la pile ou sur le tas co

Décorations d'environnement

décoration bloc =

co sur un noeud (routine) , elle indique l'ensemble des caractéristiques des paramètres (c'est l'ensemble des décorations identificateur associées aux paramètres) ce qui permet de générer le code de transmission des parties dynamiques de ces paramètres co

décoration psi =

co sur chaque noeud où il y a création d'une routine, cette décoration indique la taille de la zone des variables associée.
En la rajoutant à la taille de la zone d'évaluation, on obtient la taille de la zone statique de l'environnement associé à la routine à créer co

Décoration de modifications des valeurs

Elle correspond aux modifications (Algol 68) à apporter à la valeur retournée par le noeud qu'elle décore.

décoration modification =

co cette décoration est une liste dont chaque élément est une structure formée de trois champs :

- type de la modification Algol 68
  (dérepérer, déprocédurer, élargir, unir, ranger, neutraliser)
- mode après modification
- accès à l'élément suivant.

Décorations diverses

- décoration d'étiquette

décoration étiquette =

co représente une étiquette n'apparaissant pas dans le programme mais nécessaire pour la génération de code. Exemple : dans une instruction conditionnelle, les étiquettes permettant de "sauter" par dessus la partie alors ou la partie sinon co

- décoration de liste

décoration nbéléments =

co sur les noeuds dont un opérande est une liste, elle indique le nombre d'éléments de la liste co

- décoration d'entrées-sorties

décoration type =

co sur le noeud (entrée/sortie) , elle indique le type de l'opération à réaliser (lecture ou écriture, avec ou sans format ...etc...)

ANNEXE 2

Evaluations à partir d'un exemple

A partir d'un exemple de programme extrait du Rapport de définition du langage Algol 68 (paragraphe 11.5) nous allons donner des éléments de comparaison concernant les trois langages Algol W, PL/1, Algol 68 et leurs compilateurs, utilisables sous le système CP/CMS du calculateur IBM 360/67 du Centre Interuniversitaire de Calcul de Grenoble.

Cet exemple réalise la sommation d'une série en utilisant une transformation d'Euler convenablement affinée. La sommation est terminée lorsque l'on trouve, q fois de suite (q est égal à 2, dans l'exemple), les valeurs absolues des termes de la série transformée inférieures à une valeur EPS (cette valeur représente la précision de la sommation).

La comparaison a été effectuée en adoptant la démarche suivante :

1) - Nous avons d'abord écrit le programme en Algol 68 en y insérant une séquence servant à la lecture de la précision (EPS).

- Programme en Algol 68

```
UNITE   BLOC    TEXTE SOURCE
00000   0000    'DEBUT'
00001   0002      'PROC' EULER = ('PROC'('ENT')'REEL' F, 'REEL' EPS, 'ENT' TIM)'REEL' :
00007   0003      'DEBUT'
00008   0004        'ENT' N:=1, T; 'REEL' MN, DS:=EPS; (1:16)'REEL' M;
00015   0004        'REEL' SUM:=(M(1):=F(1))/2;
00022   0004        'POUR' I 'DEPUIS' 2 'TANT QUE' (T:=('ABS'DS<EPS | T+1 | 1)) <= TIM
00031   0008        'FAIRE'
00032   0013          MN:=F(I);
00035   0013          'POUR' K 'JUSQUA' N
00037   0013          'FAIRE'
00038   0015              MN:=((DS:=MN)+M(K))/2; M(K):=DS
00047   0015          'FAIT';
00049   0013          SUM+:=(DS:=('ABS'MN<'ABS'M(N) '&' N<16
00053   0019                    | N+:=1; M(N):=MN; MN/2
00058   0020                    | MN))
00061   0018        'FAIT';
00063   0004        SUM
00063   0004      'FIN';
00065   0002      PRINT(EULER(('ENT'I)'REEL':('ODD'I | -1/I | 1/I),
00074   0002                  ('REEL' R; READ(R); R),
00081   0002                  2))
00083   0002  'FIN'    'CO' RESULTAT = - LN(2) 'CO'
```

Une transcription fidèle de ce programme en Algol W et en PL/1 nous a fourni les programmes suivants :

- Programme en Algol W

```
0000 1-     BEGIN
0001 --           COMMENT TRANSCRIPTION EN ALGOLW;
0001 --       LONG REAL PROCEDURE EULER(LONG REAL PROCEDURE F;
0002 --                                 LONG REAL EPS;
0003 --                                 INTEGER TIM);
0004 2-       BEGIN
0005 --          INTEGER I,N,T; LONG REAL MN,DS,SUM; LONG REAL ARRAY M(1::16);
0008 --          N:=1; DS:=EPS;
0010 --          M(1):=F(1); SUM:=M(1)/2;
0012 --          I:=1;
0013 3-          WHILE BEGIN
0014 --                  T:=IF ABS(DS)<EPS THEN T+1 ELSE 1; T<=TIM
0015 -3               END
0015 3-          DO BEGIN
0016 --             I:=I+1;
0017 --             MN:=F(I);
0018 --             FOR K:=1 UNTIL N
0018 --             DO
0018 4-             BEGIN
0019 --                DS:=MN; MN:=(MN+M(K))/2; M(K):=DS;
0022 -4             END;
0023 --             DS:=IF (ABS(MN)<ABS(M(N))) AND (N<16)
0023 4-                 THEN BEGIN
0024 --                    N:=N+1; M(N):=MN; MN/2
0026 -4                 END
0026 --                 ELSE MN;
0027 --             SUM:=SUM+DS;
0028 -3          END;
0029 --          SUM
0029 -2       END; COMMENT FIN DE LA PROCEDURE EULER;
0030 --       LONG REAL PROCEDURE SERIE(INTEGER I);
0031 --          IF ODD(I) THEN -1/I ELSE 1/I;
0032 --       LONG REAL R; READ(R);
0034 --     R_W :=21; COMMENT MODIF LONGUEUR STANDARD SORTIE LONG REEL;
0035 --        WRITE(EULER(SERIE,R,2))
0035 -1     END.
```

- Programme en PL/1

```
STMT LEVEL NEST
  1                EULTEST: PROCEDURE OPTIONS(MAIN);
                          /* TRANSCRIPTION DE EULER EN PL/1 */
  2     1             DECLARE EULER ENTRY(ENTRY,FLOAT BIN(53),FIXED BIN(31))
                              RETURNS(FLOAT BIN(53));
  3     1             EULER: PROCEDURE(F,EPS,TIM) RECURSIVE RETURNS(FLOAT BIN(53));
  4     2                DECLARE F ENTRY(FIXED BIN(31)) RETURNS(FLOAT BIN(53)),
                                 EPS FLOAT BIN(53),
                                 TIM FIXED BIN(31);
  5     2                DECLARE (I, N, T) FIXED BIN(31),
                                 (MN, DS, SUM) FLOAT BIN(53),
                                 M(1:16) FLOAT BIN(53);
  6     2                N=1; DS=EPS;
  8     2                M(1)=F(1); SUM=M(1)/2;
 10     2                I=1; T=1;
 12     2                DO WHILE(T<=TIM);
 13     2     1             I=I+1;
 14     2     1             MN=F(I);
 15     2     1             DO K=1 TO N;
 16     2     2                DS=MN; MN=(MN+M(K))/2; M(K)=DS;
 19     2     2             END;
 20     2     1             IF (ABS(MN)<ABS(M(N))) & (N<16)
 21     2     1             THEN DO;
 22     2     2                N=N+1; M(N)=MN; DS=MN/2;
 25     2     2             END;
 26     2     1             ELSE DS=MN;
 27     2     1             SUM=SUM+DS;
 28     2     1             IF ABS(DS)<EPS THEN T=T+1; ELSE T=1;
 31     2     1          END;
 32     2                RETURN(SUM);
 33     2             END EULER;
 34     1             DECLARE SERIE ENTRY(FIXED BIN(31)) RETURNS(FLOAT BIN(53));
 35     1             SERIE: PROCEDURE(I) RECURSIVE RETURNS(FLOAT BIN(53));
 36     2                DECLARE I FIXED BIN(31); DECLARE RET FLOAT BIN(53);
 38     2                RET=I;
 39     2                IF MOD(I,2)=1   /* ODD(I)* /
 40     2                THEN RETURN(-1/RET); ELSE RETURN(1/RET);
 42     2             END SERIE;
 43     1             DECLARE X FLOAT BIN(53); DECLARE CHAINE CHAR(80);
 45     1             DISPLAY(' ') REPLY(CHAINE);
 46     1             GET STRING(CHAINE) LIST(X);
 47     1             DISPLAY(EULER(SERIE,X,2));
 48     1          END EULTEST;
```

Remarque : Il n'existe pas en PL/1 de fonction ou d'opérateur permettant de savoir si une valeur entière est paire ou impaire, ce qui nous a obligé à utiliser la fonction MOD.

Nous avons ensuite compilé ces trois programmes en ne demandant aucune trace de compilation si ce n'est l'impression du texte source.

Ceci fait, nous avons effectué de nombreuses exécutions de ces programmes pour des valeurs différentes de la précision. D'autre part, afin de montrer l'efficacité de la transformation d'Euler nous comptons le nombre d'appels de la fonction SERIE.

Signalons que pour chacun des programmes nous n'avons pas demandé de vérifier la compatibilité des indices avec les bornes, afin d'effectuer des mesures dans le contexte d'une exploitation d'un programme déjà mis au point.

Les résultats que nous avons obtenus sont résumés dans les tableaux suivants :

	Algol W	PL1	Algol 68
Temps de compilation (millisecondes)	360	4 350	780
Taille du code engendré (octets)	1 250	1 080	1 100
Temps de chargement (millisecondes)	200	900	75

Temps d'exécution (en millisecondes) en fonction de la valeur de EPS (1.0 e-5 dans l'exemple précédent)

EPS \ Compilateurs	Algol W	PL1	Algol 68	Nombre d'appels de SERIE
1.0 e-18	60	90	50	47
1.0 e-19	70	100	55	55
1.0 e-20	100	140	90	89
1.0 e-21	280	360	360	272
1.0 e-22	570	700	770	556
1.0 e-23	5 100	6 340	7 370	5 166

Remarque :
Nous ne nous intéressons ici qu'aux temps de calcul et non à la valeur de la somme de la série. A partir de la précision 1.0 e-14, cette valeur ne varie plus, les trois programmes donnent : -6.931471805599450E-01 ; toutefois la méthode utilisée permet de poursuivre le calcul pour des valeurs de EPS plus petites et de mesurer des temps qui croissent de manière approximativement exponentielle.

Ces résultats étant très différents (notamment pour la précision 1.0 e-23) nous avons examiné le code engendré par chacun des compilateurs et ainsi découvert plusieurs points intéressants :

a) Le compilateur PL/1 fait notamment les optimisations suivantes :

- le tableau M n'ayant qu'une seule dimension et des bornes constantes, n'est pas (pendant l'exécution) représenté au moyen d'une partie statique (dans la zone des variables) et d'une partie dynamique (dans la zone dynamique). Ses éléments sont alloués directement dans la zone des variables. Il n'y a donc pas de représentation de la partie statique, l'origine virtuelle du tableau étant connue à la compilation. Ceci optimise chaque utilisation du tableau en évitant l'indirection correspondant au chargement de l'origine virtuelle dans un registre,

- à plusieurs endroits dans le programme, une valeur réelle est divisée par la constante entière 2. Ceci étant détectable statiquement, ce n'est pas la constante entière 2 qui est mise dans la table (des constantes) mais la constante réelle 2.0. Ceci optimise chacune de ces divisions en évitant la conversion de la valeur entière 2 en la valeur réelle correspondante (une telle optimisation serait inutile si le calculateur disposait d'instructions permettant de diviser une valeur réelle par une valeur entière).

b) Le compilateur Algol W réalise, pour le tableau M, la même optimisation que le compilateur PL/1. Par contre, il ne fait pas celle concernant la division d'une valeur réelle par une constante entière.

c) Notre compilateur ne prend pas en charge ces optimisations. Signalons que notre méthodologie permet aisément de les réaliser malgré la possibilité de définition de nouveaux modes et opérateurs en Algol 68.

2) - Afin de supprimer les effets de ces optimisations nous avons réécrit les programmes en utilisant des constantes réelles (2.0) et un tableau (M) à bornes non constantes. De plus, afin de placer les compilateurs dans les mêmes conditions, nous avons (dans tous les programmes) utilisé le calcul du modulo pour connaître la parité d'une valeur entière. Nous avons ainsi obtenu les programmes suivants :

- Programme en Algol 68

```
UNITE   BLOC    TEXTE SOURCE
00000   0000    'DEBUT'
00001   0002      'PROC' EULER = ('PROC'('ENT')'REEL' F, 'REEL' EPS, 'ENT' TIM)'REEL' :
00007   0003      'DEBUT'
00008   0004        'ENT' N:=1, T; 'REEL' MN, DS:=EPS;
00012   0004        'ENT' BI:=1, BS:=16; (BI:BS)'REEL' M;
00017   0004        'REEL' SUM:=(M( 1):=F(1))/2.0;
00024   0004        'POUR' I 'DEPUIS' 2 'TANT QUE' (T:=('ABS'DS<EPS | T+1 | 1)) <= TIM
00033   0008        'FAIRE'
00034   0013          MN:=F(I);
00037   0013          'POUR' K 'JUSQUA' N
00039   0013          'FAIRE'
00040   0015              MN:=((DS:=MN)+M(K))/2.0; M(K):=DS
00049   0015          'FAIT';
00051   0013          SUM+:=(DS:=('ABS'MN<'ABS'M(N) '&' N<16
00055   0019                     | N+:=1; M(N):=MN; MN/2.0
00060   0020                     | MN))
00063   0018        'FAIT';
00065   0004        SUM
00065   0004      'FIN';
00067   0002      PRINT(EULER(('ENT'I)'REEL':(I'MOD'2=1 | -1/I | 1/I),
00076   0002                  ('REEL' R; READ(R); R),
00083   0002                  2))
00085   0002  'FIN'    'CO' RESULTAT = - LN(2) 'CO'
```

- Programme en Algol W

```
0000 1-     BEGIN
0001 --         COMMENT TRANSCRIPTION EN ALGOLW;
0001 --       LONG REAL PROCEDURE EULER(LONG REAL PROCEDURE F;
0002 --                                 LONG REAL EPS;
0003 --                                 INTEGER TIM);
0004 2-       BEGIN
0005 --         INTEGER I,N,T; LONG REAL MN,DS,SUM;
0007 --         INTEGER BI,BS; BI:=1; BS:=16;
0010 3-         BEGIN
0011 --           LONG REAL ARRAY M(BI::BS);
0012 --           N:=1; DS:=EPS;
0014 --           M(1):=F(1); SUM:=M(1)/2.0L;
0016 --           I:=1;
0017 4-           WHILE BEGIN
0018 --                    T:=IF ABS(DS)<EPS THEN T+1 ELSE 1; T<=TIM
0019 -4                END
0019 4-           DO BEGIN
0020 --              I:=I+1;
0021 --              MN:=F(I);
0022 --              FOR K:=1 UNTIL N
0022 --              DO
0022 5-              BEGIN
0023 --                 DS:=MN; MN:=(MN+M(K))/2.0L; M(K):=DS;
0026 -5              END;
0027 --              DS:=IF (ABS(MN)<ABS(M(N))) AND (N<16)
0027 5-                  THEN BEGIN
0028 --                     N:=N+1; M(N):=MN; MN/2.0L
0030 -5                  END
0030 --                  ELSE MN;
0031 --              SUM:=SUM+DS;
0032 -4           END;
0033 --           SUM
0033 -3         END
0033 -2       END; COMMENT FIN DE LA PROCEDURE EULER;
0034 --       LONG REAL PROCEDURE SERIE(INTEGER I);
0035 --         IF I REM 2 = 1 THEN -1/I ELSE 1/I;
0036 --       LONG REAL R; READ(R);
0038 --     R_W :=21; COMMENT MODIF LONGUEUR STANDARD  SORTIE LONG REEL;
0039 --       WRITE(EULER(SERIE,R,2))
0039 -1     END.
```

- Programme en PL/1

```
STMT LEVEL NEST
  1                EULTEST: PROCEDURE OPTIONS(MAIN);
                          /* TRANSCRIPTION DE EULER EN PL/1 */
  2     1             DECLARE EULER ENTRY(ENTRY,FLOAT BIN(53),FIXED BIN(31))
                              RETURNS(FLOAT BIN(53));
  3     1             EULER: PROCEDURE(F,EPS,TIM) RECURSIVE RETURNS(FLOAT BIN(53));
  4     2                DECLARE F ENTRY(FIXED BIN(31)) RETURNS(FLOAT BIN(53)),
                                 EPS FLOAT BIN(53),
                                 TIM FIXED BIN(31);
  5     2                DECLARE (I, N, T) FIXED BIN(31),
                                 (BI,BS) FIXED BIN(31),
                                 (MN, DS, SUM) FLOAT BIN(53);
  6     2                BI=1;BS=16;
  8     2                BEGIN;
  9     3                   DECLARE M(BI:BS) FLOAT BIN(53);
 10     3                   N=1; DS=EPS;
 12     3                   M(1)=F(1); SUM=M(1)/2.0;
 14     3                   I=1; T=1;
 16     3                   DO WHILE(T<=TIM);
 17     3    1                 I=I+1;
 18     3    1                 MN=F(I);
 19     3    1                 DO K=1 TO N;
 20     3    2                    DS=MN; MN=(MN+M(K))/2.0; M(K)=DS;
 23     3    2                 END;
 24     3    1                 IF (ABS(MN)<ABS(M(N))) & (N<16)
 25     3    1                 THEN DO;
 26     3    2                    N=N+1; M(N)=MN; DS=MN/2.0;
 29     3    2                 END;
 30     3    1                 ELSE DS=MN;
 31     3    1                 SUM=SUM+DS;
 32     3    1                 IF ABS(DS)<EPS THEN T=T+1; ELSE T=1;
 35     3    1              END;
 36     3                   RETURN(SUM);
 37     3                END;
 38     2             END EULER;
 39     1             DECLARE SERIE ENTRY(FIXED BIN(31)) RETURNS(FLOAT BIN(53));
 40     1             SERIE: PROCEDURE(I) RECURSIVE RETURNS(FLOAT BIN(53));
 41     2                DECLARE I FIXED BIN(31); DECLARE RET FLOAT BIN(53);
 43     2                RET=I;
 44     2                IF MOD(I,2)=1   /* ODD(I) */
 45     2                THEN RETURN(-1/RET); ELSE RETURN(1/RET);
 47     2             END SERIE;
 48     1             DECLARE X FLOAT BIN(53); DECLARE CHAINE CHAR(80);
 50     1             DISPLAY(' ') REPLY(CHAINE);
 51     1             GET STRING(CHAINE) LIST(X);
 52     1             DISPLAY(EULER(SERIE,X,2));
 53     1          END EULTEST;
```

Nous avons alors recommencé les mêmes tests que précédemment et obtenu les résultats indiqués dans le tableau ci-dessous.

EPS \ Compilateurs	Algol W	PL1	Algol 68	nombre d'appels de SERIE
1.0 2-18	60	100	30	47
1.0 2-19	60	110	50	55
1.0 2-20	90	160	100	89
1.0 2-21	250	390	300	272
1.0 2-22	500	760	650	556
1.0 2-23	4 390	6 780	5 900	5 166

Le gain de temps est appréciable pour le compilateur Algol 68. Il correspond à la suppression des conversions.

Le programme PL/1 nécessite un peu plus de temps à cause des instructions entraînées par le chargement de l'origine virtuelle du tableau.

Le gain de temps du programme Algol W, dû à la suppression des conversions, est en partie réduit par le temps nécessaire aux chargements de l'origine virtuelle du tableau.

Signalons que pour les programmes écrits en Algol W et en PL/1 le code engendré, correspondant à l'indexation du tableau M, ne fait pas la multiplication entre l'indice et l'enjambée pour obtenir l'adresse de l'élément demandé. Ce calcul est fait au moyen d'une opération de décalage, la taille de la représentation d'une valeur réelle en double précision étant une puissance de 2. Une telle optimisation permettrait de réduire considérablement le temps nécessaire à l'exécution du programme écrit en Algol 68.

Conclusion

Les résultats que nous obtenons, pour cet exemple, confirment l'intérêt de la méthodologie que nous proposons. D'autre part, ils mettent en lumière la dualité espace-temps classique, puisque c'est le programme écrit en Algol W qui donne le code le plus volumineux mais dont l'exécution est la plus rapide.

ANNEXE 3

UN EXEMPLE DE COMPILATION, DE CHARGEMENT ET D'EXECUTION D'UN PROGRAMME ALGOL 68

Nous montrons ici comment s'effectuent la compilation, le chargement et l'exécution d'un programme Algol 68. L'exemple choisi est le programme "Euler" présenté précédemment. Nous détaillons le processus de segmentation, et nous donnons une "trace" fournie par le compilateur, qui permet de suivre pas à pas la génération de code. Nous montrons également les tables d'identificateurs et de modes, construites lors de la compilation de ce programme, le code chargé, et l'état de la pile après son exécution.

Les commandes de compilation, de chargement, et d'exécution -

- Compilation : Le programme à compiler est sur un fichier nommé EULER2. Les options de compilation demandées sont :

- arbre - édition des sous-arbres,
- code - trace du code engendré,
- trans - trace du parcours de génération.

```
algol68 euler2 ( arbre code trans

EXECUTION BEGINS...
TEMPS DE COMPILATION.          2313,93 MILLISECONDES
R; T=2.51/3.75 16:17:46
```

La compilation produit trois fichiers :

- un fichier de type SORTIE, qui contient la liste du programme, avec les éventuels messages d'erreur,
- un fichier de type EDITION, qui contient les traces demandées,
- un fichier de type STANDOBJ, qui contient le code chargeable.

- Chargement et exécution :

L'option de chargement "code", permet d'obtenir la liste du code chargé. L'option d'exécution "dump", permet d'obtenir une image de l'état de la mémoire après l'exécution du programme.

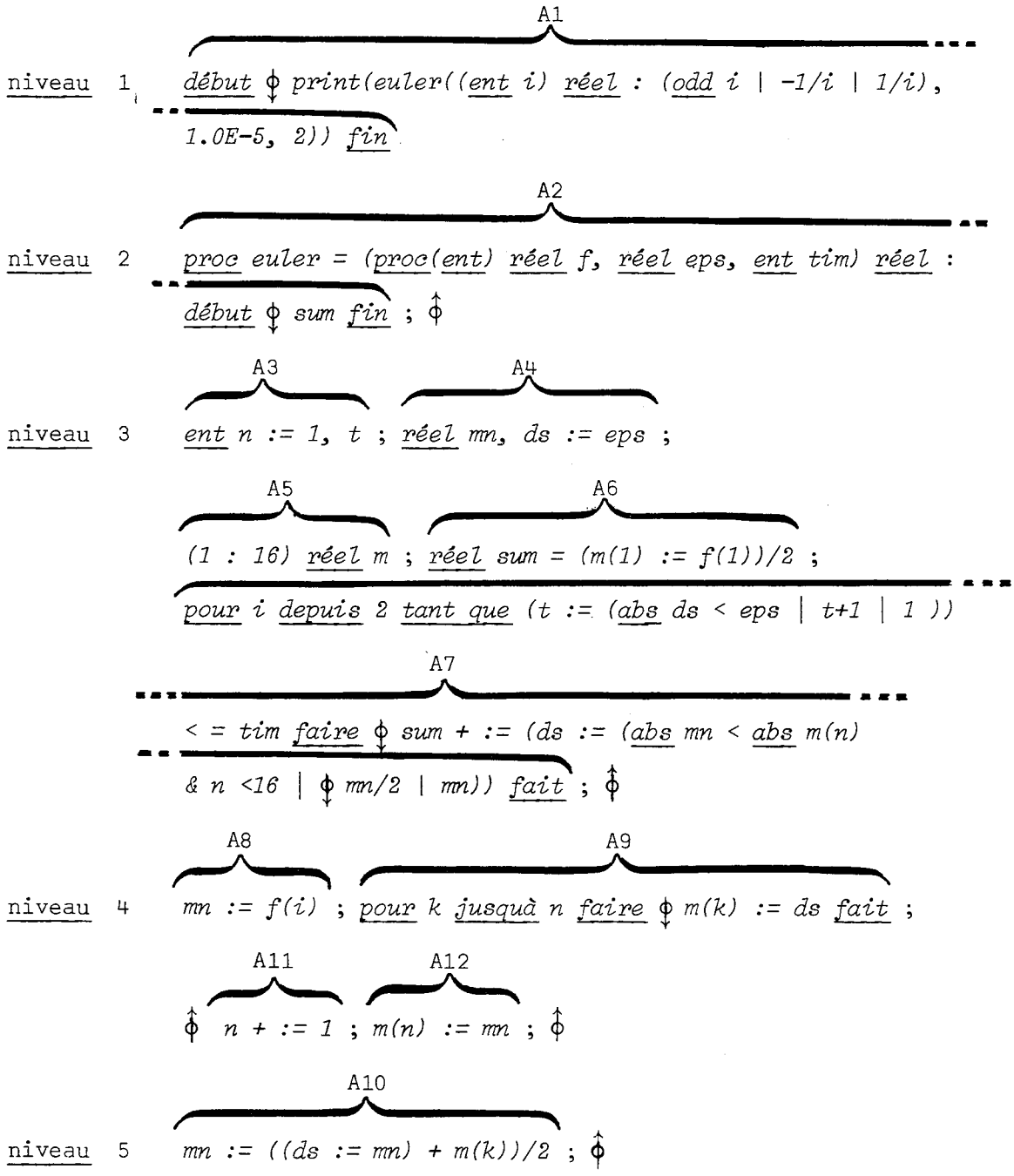
```
168 euler2 ( code dump

EXECUTION BEGINS...
     TEMPS DE CHARGEMENT       536,66 MILLISECONDES
EXECUTION LANCEE...
-.6931487024260 46E+00
     TEMPS D'EXECUTION           0,43 MILLISECONDES
R; T=0.67/1.09 16:19:56
```

le résultat de l'exécution apparaît à la console, qui est le fichier standard de sortie. Il est créé un fichier de type STANDOUT, qui contient les traces de chargement et d'exécution.

Segmentation du programme

Le texte produit par le second passage est sous forme postfixée segmentée. Nous indiquons ci-dessous le mode de découpage du programme, en numérotant les arbres correspondant aux unités de segmentation, dans l'ordre de leur création. Nous donnons une image infixée du texte postfixé, afin d'en faciliter la lecture. Nous rappelons ensuite le schéma d'imbrication des phases de construction des sous-arbres, de traitement des modifications et de génération de code pour chaque sous-arbre, puis nous donnons "en clair" l'image de chaque sous-arbre.

Le texte segmenté, par niveaux



LA LISTE COMPLETE DES OPTIONS DE COMPILATION EST --

```
                                        SOURCE
                                      NOLEXTR
                                        ARBRE
                                      NOICL
                                        CODE
                                      NOCODBUF
                                      NODET
                                      NODESCR
                                        P2
                                        P3
                                      NOTRACEP1
                                      NOTRACEP2
                                      NODEBUG
                                      NOEXECM
```



```
                UNITE   BLOC    TEXTE SOURCE

                00000   0000    'DEBUT'
                00001   0002      'PROC' EULER = ('PROC'('ENT')'REEL' F, 'REEL' EPS, 'ENT' TIM)'REEL' :
                00007   0003      'DEBUT'
                00008   0004        'ENT' N:=1, I; 'REEL' MN, DS:=EPS; (1:16)'REEL' M;
                00015   0004        'REEL' SUM:=(M(1):=F(1))/2;
                00022   0004        'POUR' I 'DEPUIS' 2 'TANT QUE' (I:=('ABS'DS<EPS | I+1 | 1)) <= TIM
                00031   0008        'FAIRE'
                00032   0013          MN:=F(I);
                00035   0013          'POUR' K 'JUSQUA' N
                00037   0013          'FAIRE'
                00038   0015              MN:=((DS:=MN)+M(K))/2; M(K):=DS
                00047   0015          'FAIT';
                00049   0013          SUM+:=(DS:=('ABS'MN<'ABS'M(N) '&' N<16
                00053   0019                    | N+:=1; M(N):=MN; MN/2
                00058   0020                    | MN))
                00061   0018        'FAIT';
                00063   0004        SUM
                00063   0004      'FIN';
                00065   0002      PRINT(EULER(('ENT'I)'REEL':('ODD'I | -1/I | 1/I),
                00074   0002                  1.0E-5,
                00075   0002                  2))
                00077   0002  'FIN'    'CC' RESULTAT = - LN(2) 'CC'
```

TEMPS DE COMPILATION. 2313,93MILLISECONDES

TEMPS DE RESIDENCE. 20.00 SECONDES.

B. BRETAGNOLLE, P.Y. CUNIN, J. VOIRON

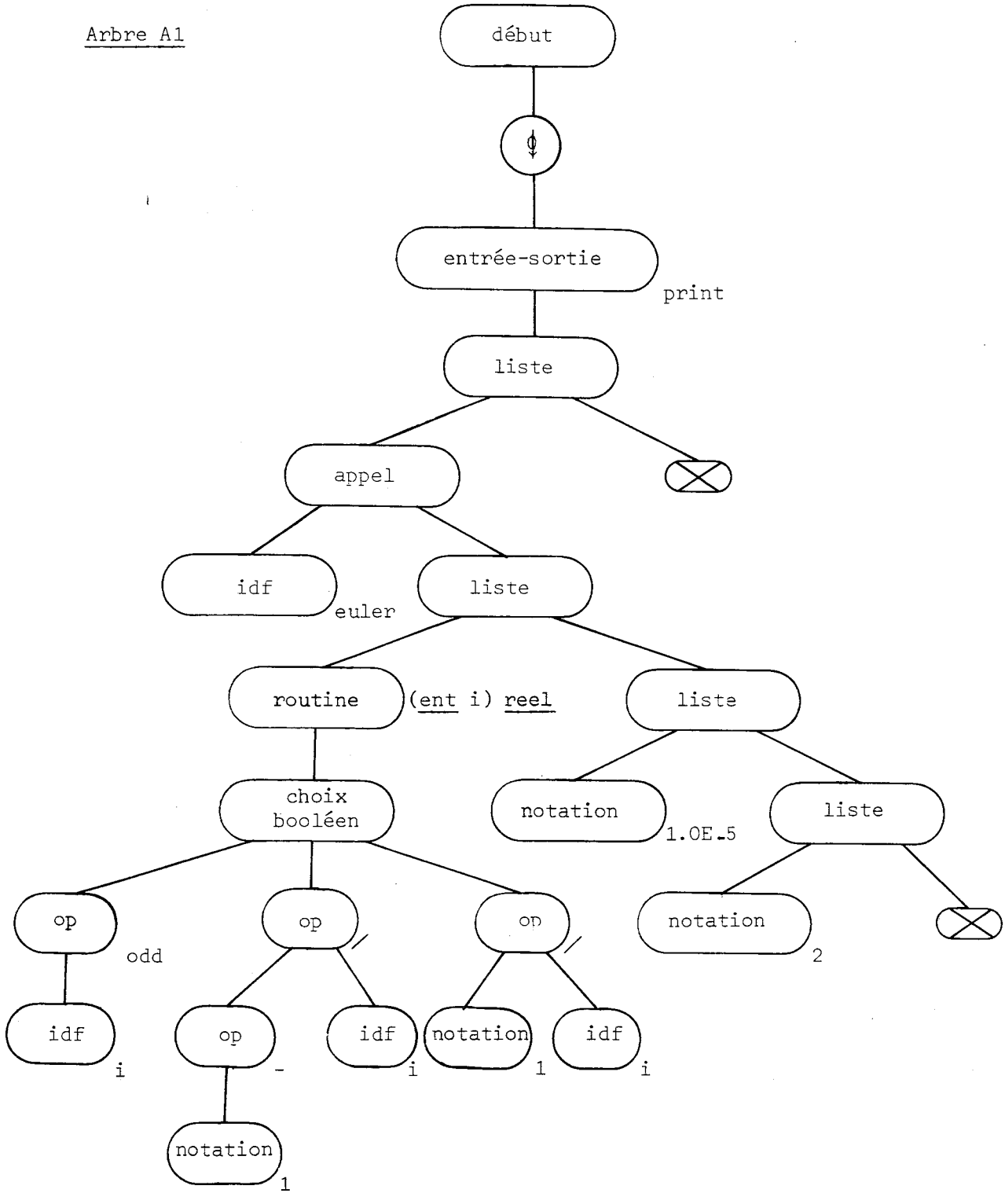
Imbrication des phases de traitement des modifications et de génération de code

	Arbres Présents en mémoire
Construction de l'arbre A1	A1
Traitement des modifications (A1)	
Génération de code (A1)	
↓	
Construction de l'arbre A2	A1 A2
Traitement des modifications (A2)	
Génération de code (A2)	
↓	
Construction de l'arbre A3	A1 A2 A3
Traitement des modifications (A3)	
Génération de code (A3)	
Destruction de l'arbre A3	
Construction de l'arbre A4	A1 A2 A4
Traitement des modifications (A4)	
Génération de code (A4)	
Destruction de l'arbre A4	
Construction de l'arbre A5	A1 A2 A5
Traitement des modifications (A5)	
Génération de code (A5)	
Destruction de l'arbre A5	
Construction de l'arbre A6	A1 A2 A6
Traitement des modifications (A6)	
Génération de code (A6)	
Destruction de l'arbre A6	
Construction de l'arbre A7	A1 A2 A7
Traitement des modifications (A7)	
Génération de code (A7)	
↓	
Construction de l'arbre A8	A1 A2 A7 A8
Traitement des modifications (A8)	
Génération de code (A8)	
Destruction de l'arbre A8	

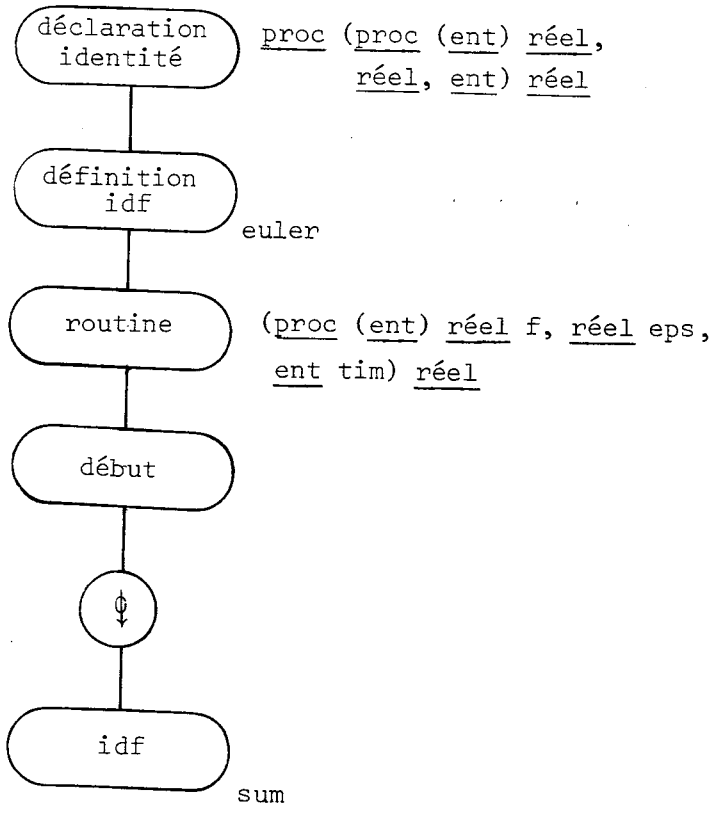
Construction de l'arbre A9	A1 A2 A7 A9
Traitement des modifications (A9)	
Génération de code (A9)	
↓	
Construction de l'arbre A10	A1 A2 A7 A9 A10
Traitement des modifications (A10)	
Génération de code (A10)	
Destruction de l'arbre A10	
↑	
Destruction de l'arbre A9	A1 A2 A7
↑	
...	
↓	
Création de l'arbre A11	A1 A2 A7 A11
Traitement des modifications (A11)	
Génération de code (A11)	
Destruction de l'arbre A11	
Création de l'arbre A12	A1 A2 A7 A12
Traitement des modifications (A12)	
Génération de code (A12)	
Destruction de l'arbre A12	
↑	
Destruction de l'arbre A7	A1 A2
↑	
Destruction de l'arbre A2	A1
↑	
Destruction de l'arbre A1	

Nous donnons maintenant l'image de chaque sous-arbre produit.

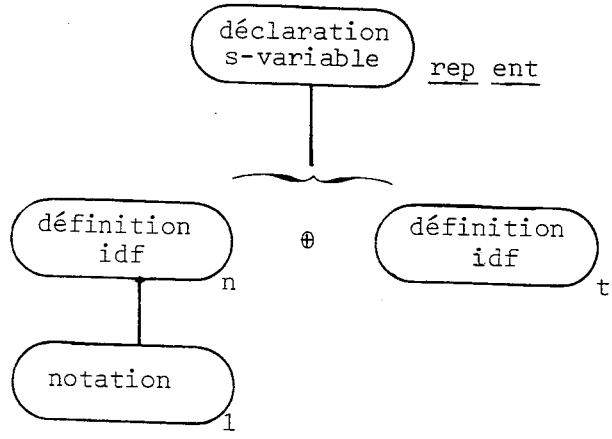
Arbre A1


début
entrée-sortie
print
liste
appel
idf
euler
liste
routine
(ent i) reel
liste
choix booléen
notation
1.0E-5
liste
op
odd
op
op
notation
2
idf
i
op
-
idf
i
notation
1
idf
i
notation
1

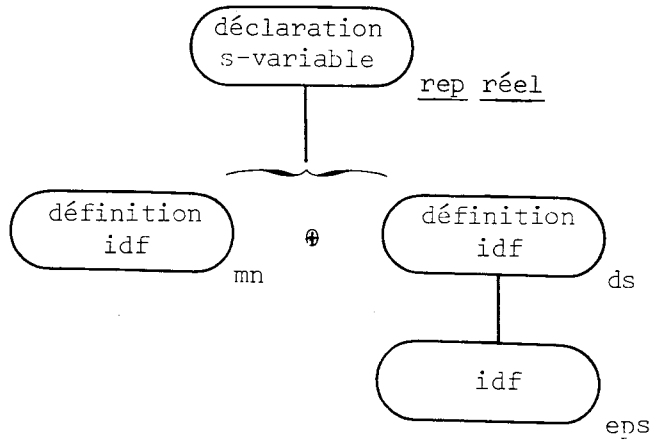
Arbre A2



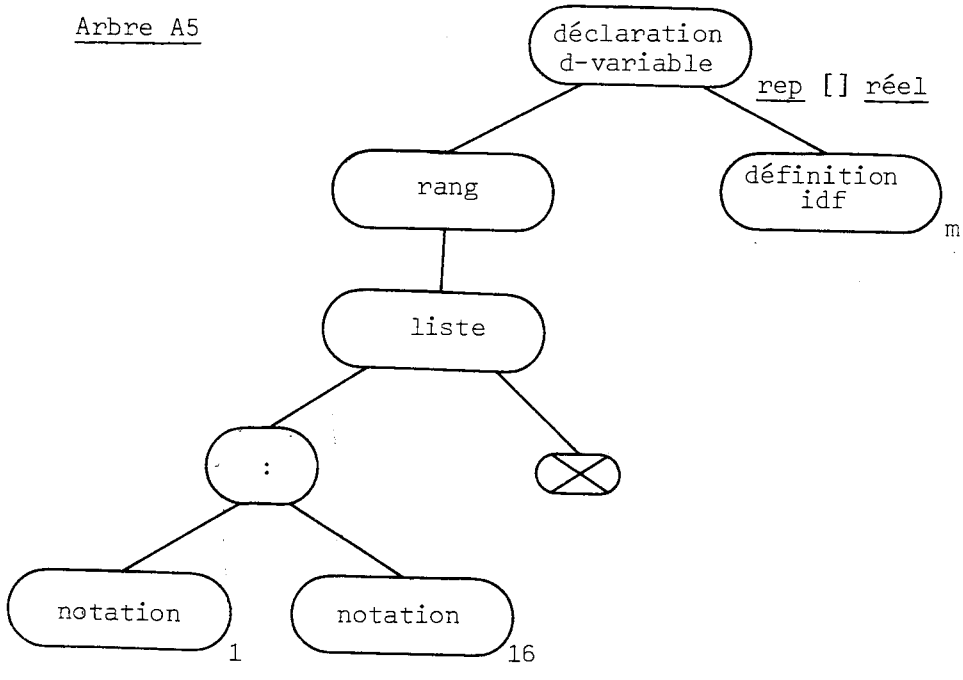
Arbre A3



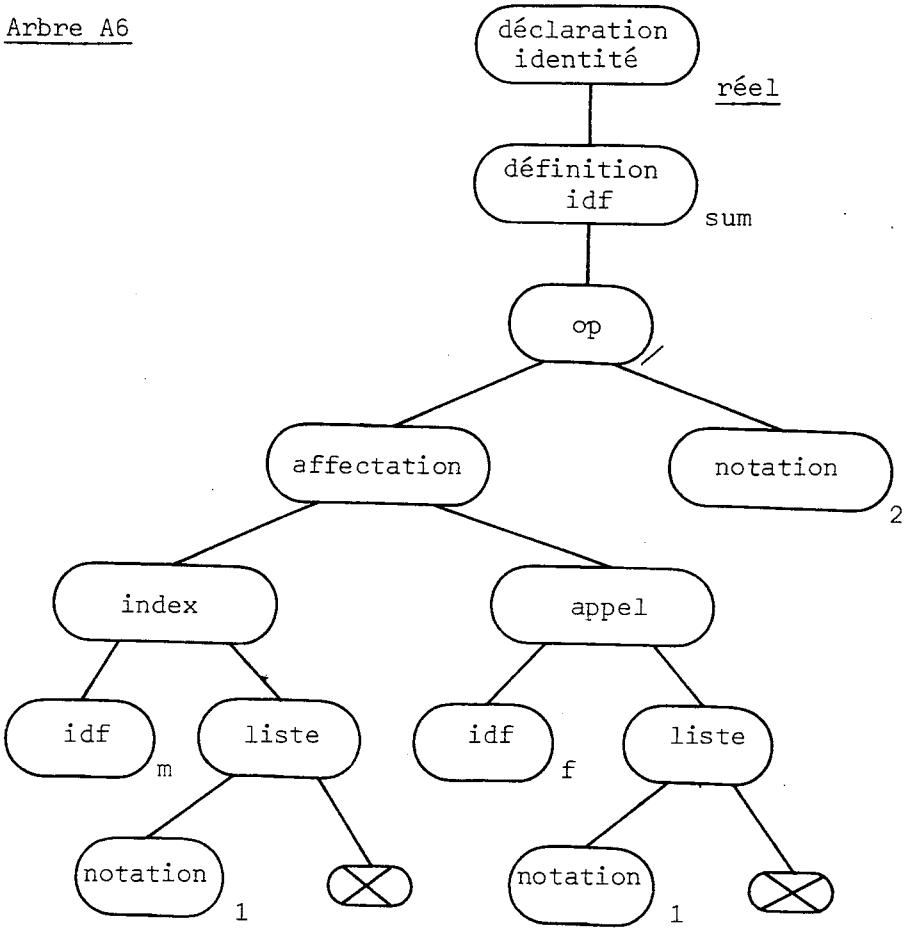
Arbre A4



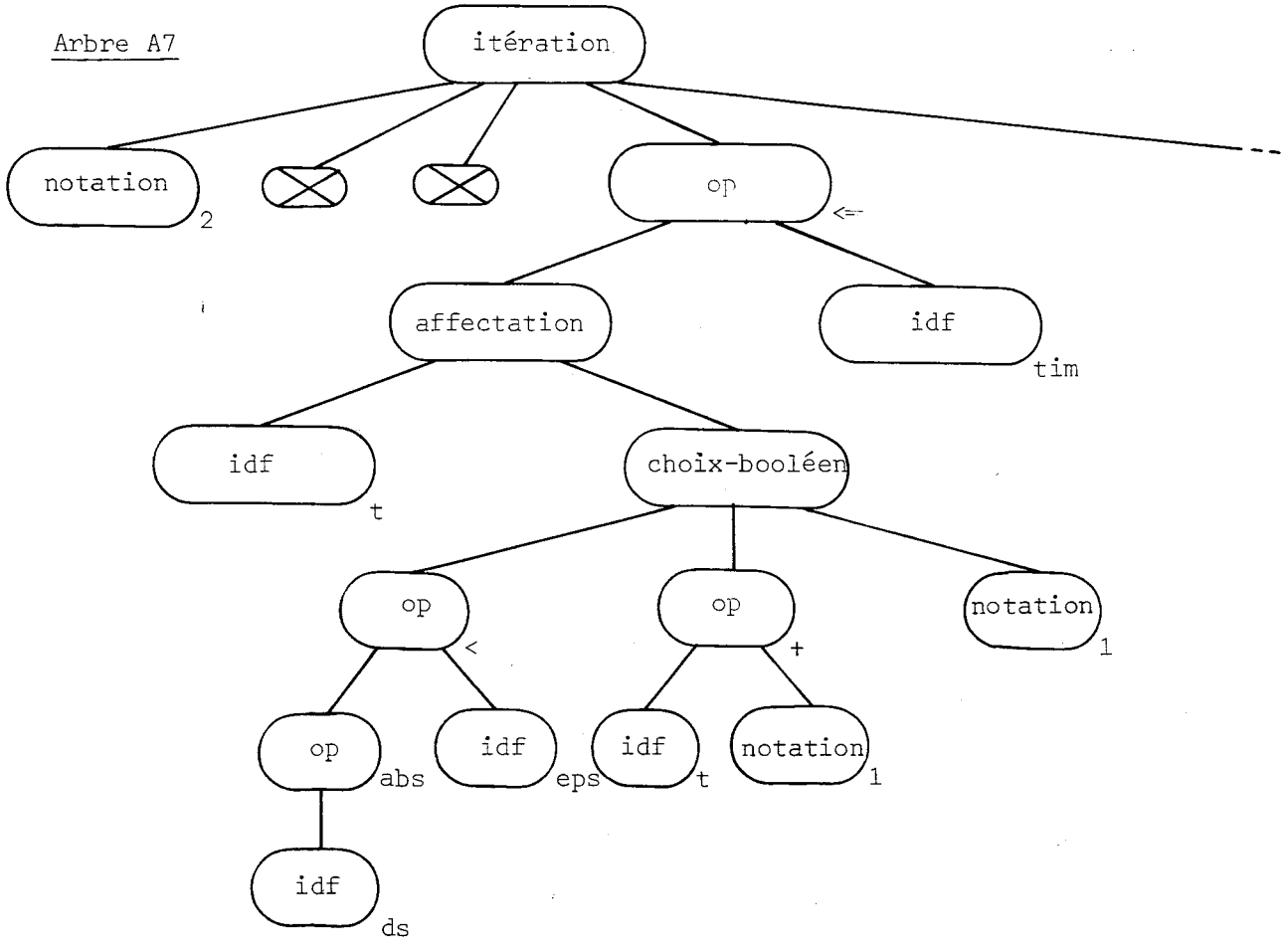
Arbre A5

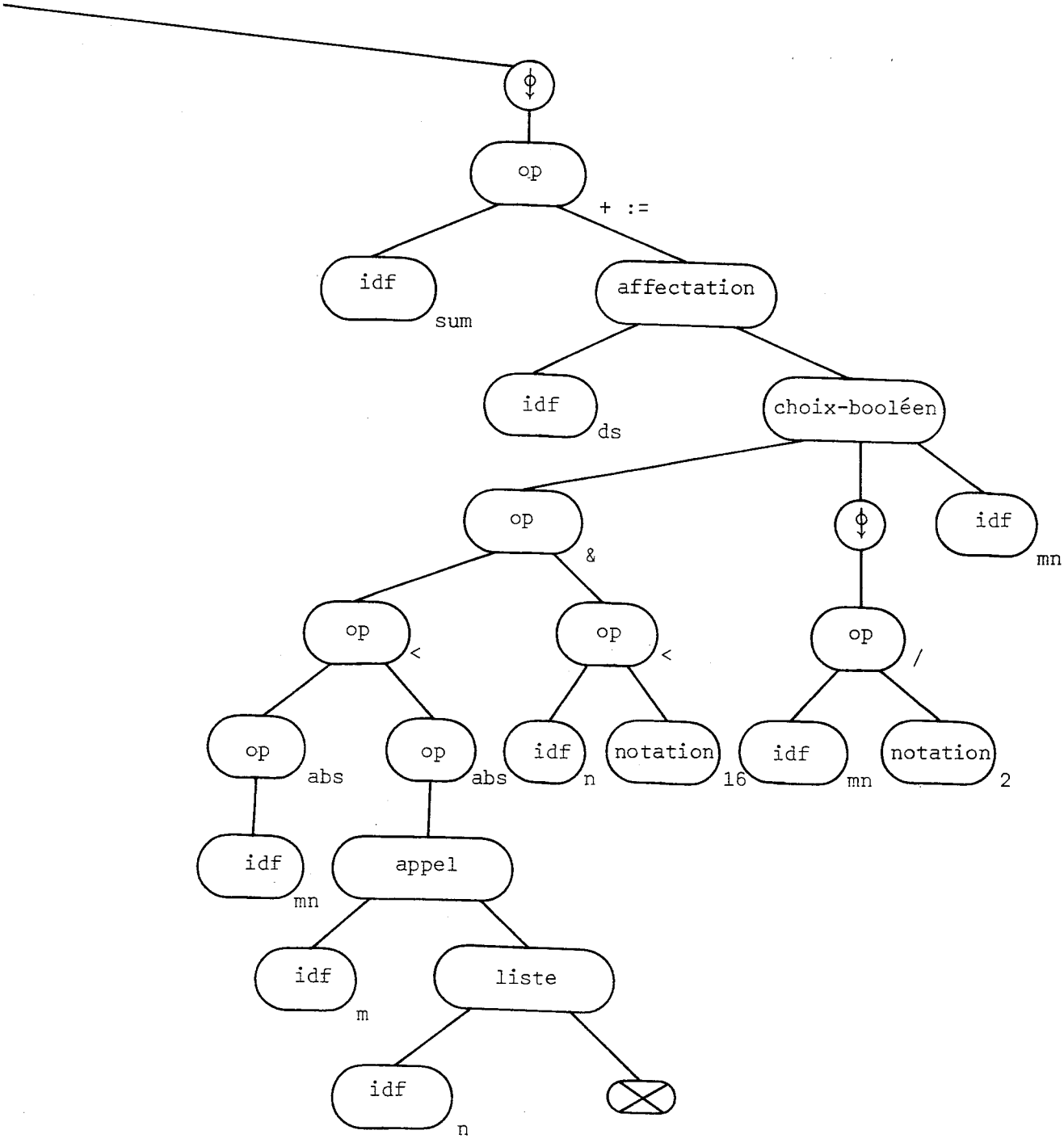


Arbre A6

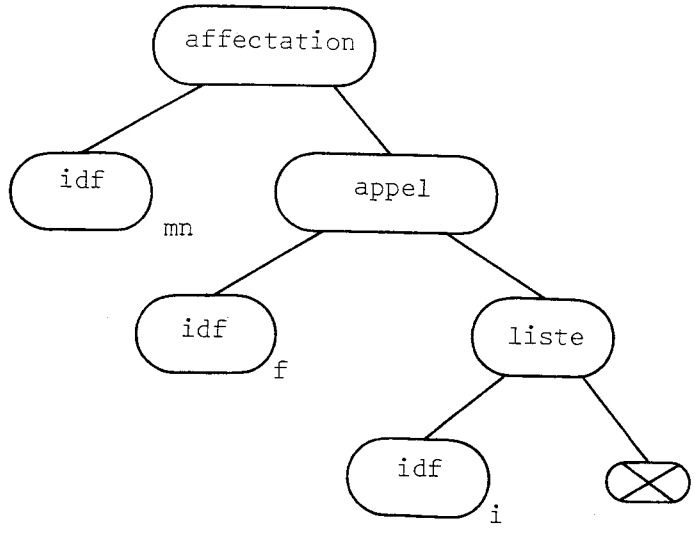

déclaration identité
réel
définition idf
sum
op
affectation
notation
2
index
appel
idf
m
liste
idf
f
liste
notation
1
notation
1

Arbre A7

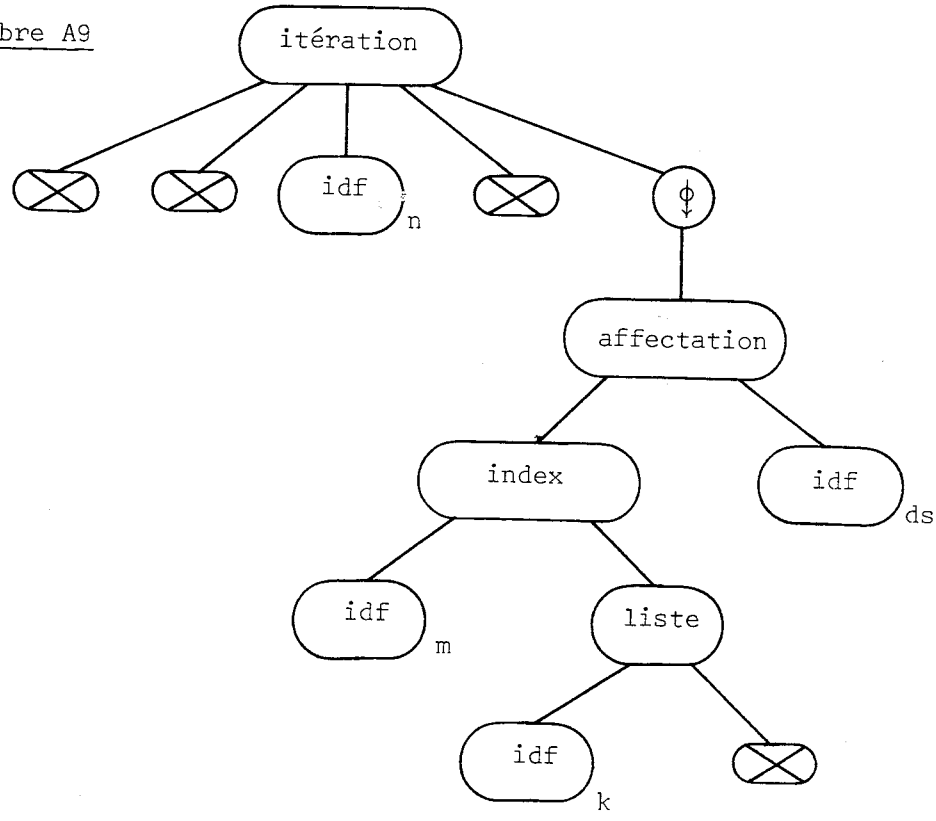

itération
notation
2
op
<=-
affectation
idf
tim
idf
t
choix-booléen
op
<
op
+
notation
1
op
abs
idf
eps
idf
t
notation
1
idf
ds


op
+ :=
idf
sum
affectation
idf
ds
choix-booléen
op
&
op
<
op
<
op
/
idf
mn
op
abs
op
abs
idf
n
notation
16
idf
mn
notation
2
idf
appel
idf
m
liste
idf
n

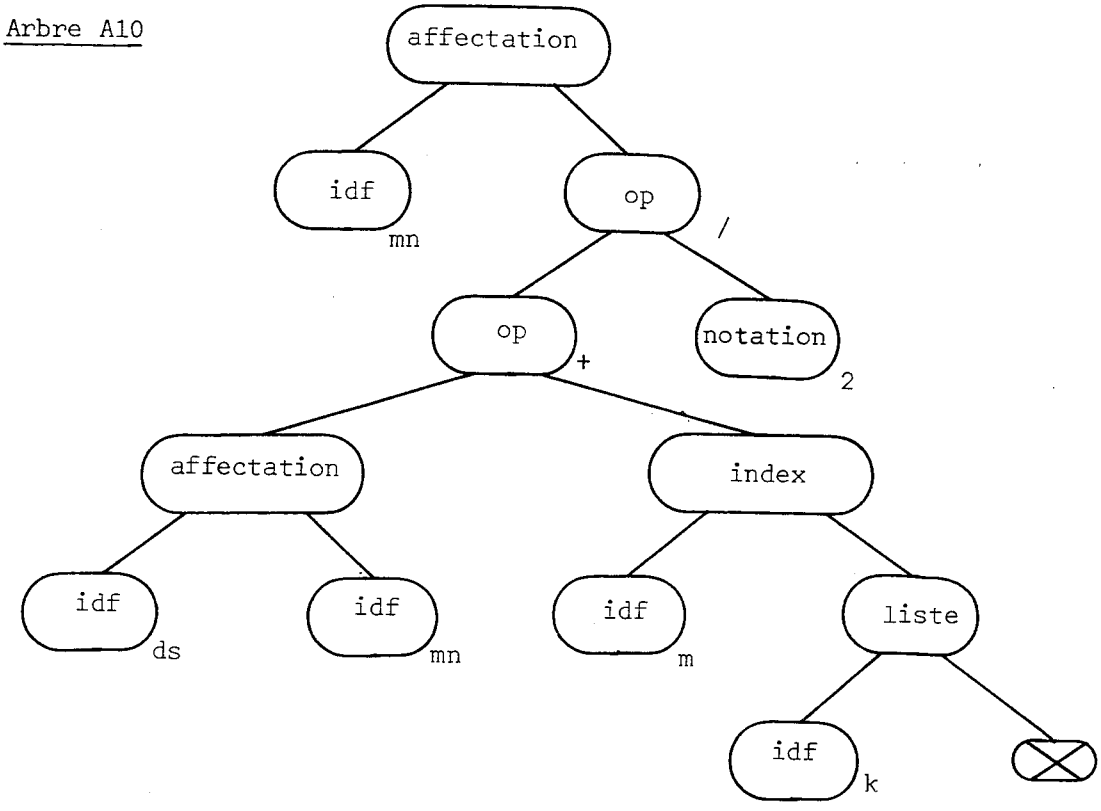
Arbre A8



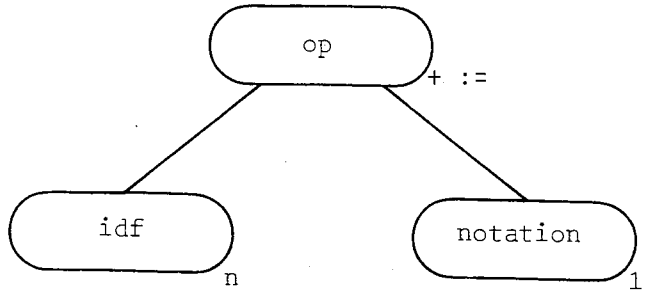
Arbre A9



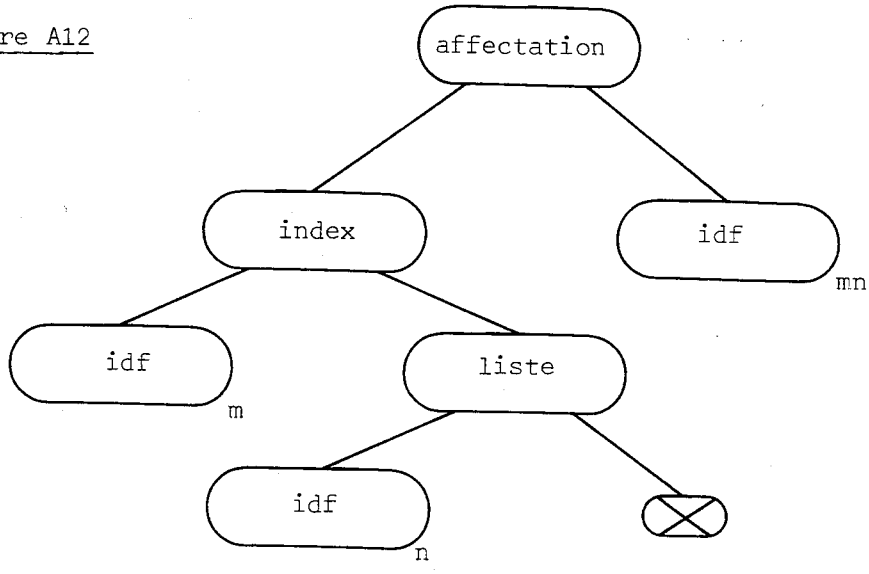
Arbre A10


affectation
idf
mn
op
/
op
+
notation
2
affectation
index
idf
ds
idf
mn
idf
m
liste
idf
k

Arbre A11



Arbre A12



La trace de génération de code

Nous donnons dans les notices techniques, les caractéristiques détaillées de tous les noeuds de l'arbre [A68TECH]. Il n'est pas nécessaire de connaître en détail les décorations de chaque noeud, pour comprendre l'organisation du traitement des modifications et de la génération de code, ce qui est le but visé ici. Nous donnons à titre d'exemple le format de trois noeuds : le noeud (début) , le noeud (appel) , et le noeud (idf) , tels qu'ils apparaissent dans l'exemple présenté, le format des tables et celui du code engendré.

format des noeuds

Le format d'un noeud tel qu'il apparaît sur la liste est le suivant :

adresse hexadécimale du noeud, type du noeud, numéro d'unité, adresse(s) du (des) opérande(s) s'il(s) existe(nt), décorations, liste des modifications attachées à ce noeud.

Exemples :

1) - Le noeud (début)

```
* 087F54   BEGIN   1     * 087F60       2   4
                          PAS DE CLIST
```

Le noeud BEGIN ((début)) se trouve à l'adresse 087F54, son numéro d'unité est 1, et il a un opérande, qui se trouve à l'adresse 087F60 ; ses décorations sont les suivantes :

2 : numéro caractéristique de la région
4 : numéro du mode contextuel (neutre)

Ce noeud n'a pas de décoration modification, ce qui est rappelé par : PAS DE CLIST.

2) - Le noeud (appel)

```
*087F9C   CALL   67   *088150      *087FB8      *NIL
                  6  9- 3   0      3    0       255
                 CLIST VIDE
```

Le noeud CALL ((appel)) se trouve à l'adresse hexadécimale 087F9C, son numéro d'unité est 67, il a deux opérandes, qui sont aux adresses respectives 088150 et 087FB8 ; ses décorations sont les suivantes :

NIL : liste de modifications vide
6 : numéro du mode propre (réel)
9- : numéro du mode contextuel (éditable)
3 : nombre de paramètres
0
3
0
255
} décorations temporaires utilisées dans les phases de décoration de l'arbre

La liste de modifications étant vide, il est indiqué CLIST VIDE

3) - Le noeud (idf)

```
*087F10    IDF     63        *087EB4
                   261       6     4     60
                   DEREF     6
```

Le noeud IDF ((idf)) se trouve à l'adresse 087F10, son numéro d'unité est 63, et il n'a aucun opérande ; ses décorations sont les suivantes :

087EB4 adresse de la liste de modifications qui lui est attachée,

261 : numéro du mode propre (rep réel)
6 : numéro du mode contextuel (réel)
4 } accès à l'enregistrement associé à la déclaration (numéro
60 } de la région et déplacement dans la table des identificateurs)

La liste de modifications est constituée d'un dérepérage DEREF qui fournit le mode réel, dont le numéro de mode est 6.

Format des tables (en décimal)

1) - La table des parenthèses (table des blocs) voir définition chap. III.4, §. 1.1.4, et exemple p.574.

Elle contient, pour chaque région les caractéristiques de cette région et l'accès aux différentes tables de la région :

ADDTIM	adresse de la table des indicateurs de mode, ici -1029, c'est-à-dire nil (table inexistante)
ADDTIP	adresse de la table des indicateurs de priorité
ADDTID	adresse de la table des identificateurs
ADDTOP	adresse de la table des opérateurs.

La table des identificateurs contient les informations suivantes :

NUMIDTID	numéro lexicographique de l'identificateur
CLASSTID	numéro de classe du mode de l'identificateur
DEPLTID	déplacement de l'identificateur dans la zone des variables, à l'exécution
STOUET	déclaration stricte ou étendue.

2) - La table des classes - voir définition chap. III.4, § 1.1.1,
et exemple p. 575.

Cette table contient les caractéristiques statiques des modes introduits dans le programme. Un certain nombre de modes, dont les modes standard, ont des numéros de classe fixés. Pour chaque classe on trouve :

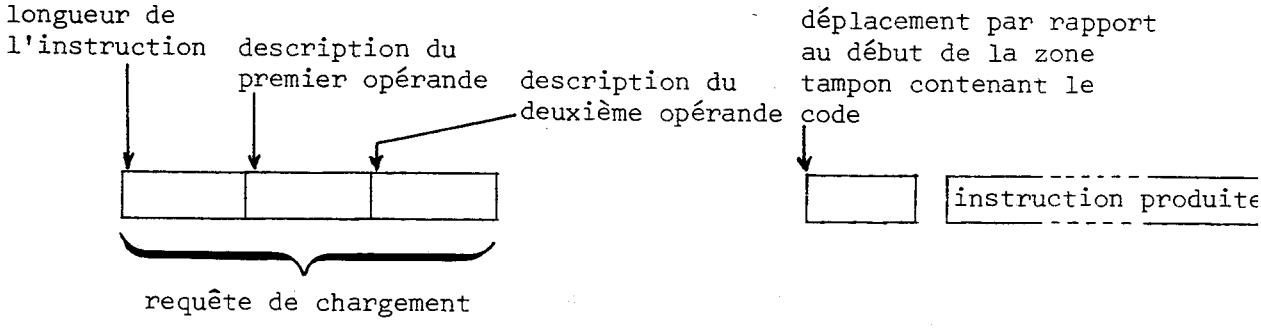
- le numéro de la classe
- STATTAILLE taille de la partie statique
- ALIGNEMENT facteur d'alignement
  etc...

chaque mode est formé de :

- un code identifiant le type du déclareur (exemple CODEREF)
- le numéro de classe du déclareur
- le numéro de classe des sous-déclareurs (exemple : SSDECREF)

Format du code (en hexadécimal)

Chaque instruction apparaît sous la forme :



Exemples :

1) - 06 00 00 012C 58 19 0 0018

La taille de l'instruction engendrée est 06, ce qui comprend la requête de chargement, et l'instruction L R1, 24(R9). Cette instruction se trouve au déplacement 12C dans la zone tampon.

2) - 08 00 19 013E 98 EF 9 0030

L'instruction correspondante en assembleur est : LM R14, R15, 48(R9). Le champ de la requête de chargement, qui décrit le deuxième opérande, contient la valeur 19, ce qui signifie que le deuxième opérande de l'instruction - ici 48(R9) - se trouve dans la zone d'évaluation locale dont le registre de base est R9.

La trace de génération

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

ARBRE ABSTRAIT DECORE **A1**

*C87F54	BEGIN	1	*C87F60		2	4						
			PAS DE CLIST									
*C87F6C	DOWN	0	*C87F70		2	2	0	255				
			PAS DE CLIST									
*C87F7C	ENTSORT	66		*C87F8C	*NIL	4	4	1	0	5	0	255
			CLIST VIDE									
*C87F8C	COMMAIND	0	*C87F9C	*NIL		9-	0					
			PAS DE CLIST									
*C87F9C	CALL	67	*C88150	*C87F88	*NIL	6	9-	3	0	3	0	255
			CLIST VIDE									
*C87F88	COMMAIND	0	*C88008	*C87FC8		316	1					
			PAS DE CLIST									
*C87FC8	COMMAIND	0	*C87FF8	*C87FC8		316	2					
			PAS DE CLIST									
*C87FD8	COMMAIND	0	*C87FE8	*NIL		316	3					
			PAS DE CLIST									
*C87FE8	DENOT	75	*NIL		10	10	26					
			CLIST VIDE									
*C87FF8	DENOT	74	*NIL		6	6	38					
			CLIST VIDE									
*C88008	ROUTINE	68	*C88024	*NIL		315	315	5	0	48	22	255
			CLIST VIDE									
*C88024	IFTE	70	*C88114	*C8809C	*C88050	*C30B6E	1	256-	0	6	5	24

			25	23	71	72	255					
			PAS DE CLIST									
*088050	OP	72	*08808C	*08807C	*NIL	6	6	1	444	55	1	444
			161	0	0	0	5	255	0			
			CLIST VIDE									
*08807C	IDF	72	*NIL	10	34	22	0					
			CLIST VIDE									
*08808C	DENOT	72	*NIL	10	10	2						
			CLIST VIDE									
*0880SC	OP	71	*0880D8	*0880C8	*NIL	6	6	1	444	55	1	444
			161	0	0	0	5	255	0			
			CLIST VIDE									
*0880C8	IDF	71	*NIL	10	34	22	0					
			CLIST VIDE									
*0880D8	OP	71	*088104	*NIL	*NIL	10	10	1	420	181	1	420
			67	0	0	256-	4	255	0			
			CLIST VIDE									
*088104	DENOT	71	*NIL	10	10	2						
			CLIST VIDE									
*088114	OP	70	*088140	*NIL	*NIL	13	13	1	492	270	1	492
			71	0	0	0	3	255	0			
			CLIST VIDE									
*088140	IDF	70	*NIL	10	9	22	0					
			CLIST VIDE									
*088150	IDF	66	*NIL	316	4-	2	0					
			CLIST VIDE									

****

GEN DESCENTE DANS NOEUD BEGIN

04 00 16 0000 0001 ****

GEN DESCENTE DANS NOEUD DOWN

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

ARBRE ABSTRAIT DECORE **A2**

```
*087EBC  IDTY          1    *C87EC8      1      316
                           PAS DE CLIST

*087EC8  IDFDEF        1    *C87EC8   *NIL       2        0
                           PAS DE CLIST

*087ED8  ROUTINE       2    *C87EF4   *NIL     316      316        5        0      144        3      255
                           CLIST VIDE

*C87EF4  BEGIN         8    *C87F00      4       6
                           PAS DE CLIST

*C87F0C  DOWN          0    *C87F10      4       3        0      255
                           PAS DE CLIST

*C87F1C  IDF          63    *C87EB4    261       6        4       60
                           DEREF        6
```

GEN DESCENTE DANS NOEUD IDTY
GEN DESCENTE DANS NOEUD ROUTINE

```
                                   04 00 16  0004  0002                          ****
                                   04 00 02  0000                        CL.2600 EQU  *
                                   04 00 07  0004  0002                          ****
                                   06 00 19  0008  41 79 0 0090                  LA
                                   00 00 01  000E  5879001402037000E000          TAGC   21
                                                   58F011AA4120D1804140
                                                   58F0
GEN DESCENTE DANS NOEUD  BEGIN
GEN DESCENTE DANS NOEUD  DOWN      04 00 16  0028  0008                          ****
```

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

****

ARBRE ABSTRAIT DECORE **A3**

****

```
*087ECC  VAR          8    *087E1C      10        2       255
                           PAS DE CLIST

*087E1C  IDFDEF       8    *087E3C  *087E2C      4         0
                           PAS DE CLIST

*087E2C  IDFDEF       9    *NIL     *NIL         4        12
                           PAS DE CLIST

*087E3C  DENOT        8    *NIL        10        10         2
                           CLIST VIDE
```

****

```
GEN DESCENTE DANS NOEUD  VAR
GEN DESCENTE DANS NOEUD  DENOT
GEN REMONTER DANS NOEUD  IDFDEF
                                        06 00 21  0020  58 6B 0 0000        L
                                        06 00 19  0032  50 69 0 0058        ST
GEN REMONTER DANS NOEUD  DOWN
```

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

****

ARBRE ABSTRAIT DECORE **A4**

****

```
*C87ECC  VAR          10    *C87E1C        6        2      255
                          PAS DE CLIST

*C87E1C  IDFDEF       10    *NIL      *087E2C      4       24
                          PAS DE CLIST

*C87E2C  IDFDEF       11    *C87E3C   *NIL         4       36
                          PAS DE CLIST

*C87E3C  IDF          11    *NIL           6       6        3       12
                          CLIST VIDE
```

****

```
GEN DESCENTE DANS NOEUD  VAR                    04 00 16  0038  000A          ****
GEN DESCENTE DANS NOEUD  IDF                    04 00 16  003C  000B          ****
GEN REMONTER DANS NOEUD  IDFDEF                 12 19 19  0040  D2 07 9 0048 9 0028  MVC
GEN REMONTER DANS NOEUD  DOWN
```

```
GRENOBLE ALGOL 68.                                                                    01 DECEMBRE 1975 @ 16:172400  PAGE   6

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION
 *************************************************************************************************************************

        ARBRE ABSTRAIT DECORE   A5

 *************************************************************************************************************************

*CE7DEC  ROWVAR        13     *C87E04   *087CF4      40        317          1         255
                             PAS DE CLIST

*CE7CF4  IDFDEF        14     *NIL      *NIL          4         48
                             PAS DE CLIST

*CE7.C4  ROW           13     *C87E14   *NIL          1          1
                             PAS DE CLIST

*CE7E14  COMMALIS       0     *C87E20   *NIL
                             PAS DE CLIST

*CE7-2C  COLON         13     *C87E3C   *087E2C
                             PAS DE CLIST

*CE7E2C  DENOT         13     *NIL           10        10         14
                             CLIST VIDE

*CE7E3C  DENOT         13     *NIL           10        10          2
                             CLIST VIDE
*************************************************************************************************************************
GEN DESCENTE DANS NOEUD  RCWVAR
                                            04 00 16  004C  000C                                ****
                                            06 00 19  0050  41 29 0 0098                        LA
                                            06 00 19  0056  41 50 0 0001                        LA
                                            04 00 02  0000                          CL.2001     LWU    *
                                            04 00 07  0004  000D                                ****
                                            06 00 19  0008  41 79 0 0028                        LA
                                            00 00 01  000E  50790014D20770J0E000                TAGC     21
                                                            58F0114A41200180414O
                                                            58F0
                                            06 00 19  0028  50 29 0 0038                        ST
                                            04 00 00  002E  18 00                               SR
                                            06 00 19  0032  50 59 0 003C                        ST
                                            06 00 21  0038  58 08 0 0064                        L
                                            06 00 19  003E  50 02 0 0000                        ST
                                            06 00 19  0044  50 52 0 0004                        ST
                                            06 00 21  004A  58 08 0 0008                        L
                                            06 00 19  0050  50 02 0 0014                        ST
                                            06 00 21  0056  58 08 0 000C                        L
                                            06 00 19  005C  50 02 0 0018                        ST
```

```
GEN DESCENTE DANS NOEUD  DENCT

GEN REMONTER DANS NOEUD  COLCN


GEN DESCENTE DANS NOEUD  DENCT
GEN REMONTER DANS NOEUD  COLCN


GEN REMONTER DANS NOEUD  DCHN
```

```
04 00 16  0062  0006                    ****

06 00 21  0066  58 0B 0 0000            L
06 00 19  006C  58 29 0 0038            L
06 00 19  0072  50 02 0 001C            ST

06 00 21  0076  58 0B 0 0010            L
06 00 19  007E  58 29 0 0038            L
06 00 19  0084  50 02 0 0020            ST
06 00 19  008A  41 22 0 0008            LA
06 00 03  0090  58 FC 0 09CE            L
04 00 00  0096  05 1F                   BALR
04 00 00  009A  0003                    ****
06 00 00  009E  98 EF 0 9008            LM
06 00 19  00A4  58 29 0 0038            L
06 00 19  00AA  58 52 0 0004            L
04 00 00  00B0  18 75                   LR
06 00 19  00B4  5C 62 0 000C            M
06 00 19  00BA  5A 79 0 0030            A
00 00 01  00C0  417070078870000389 70   TAGC    11
                58F0
06 00 19  00D0  50 79 0 0030            ST
06 00 19  00D6  58 29 0 0030            L
04 00 08  00DC  0040                    ****
06 00 03  00E0  58 FC 0 0A29            L
04 00 00  00E6  18 EC                   LR
04 00 00  00E8  05 1F                   BALR
00 00 01  00EA  98EF9008                TAGC    03
06 00 19  00F2  41 30 0 0001            LA
06 00 03  00F8  58 FC 0 09CE            L
04 00 00  00FE  05 1F                   BALR
04 00 00  0102  0004                    ****
06 00 00  0106  98 EF 0 9008            LM
04 00 00  010C  18 53                   LR
06 00 19  0110  41 35 0 0060            LA
04 00 00  0116  1B 66                   SR
04 00 00  0118  1B 77                   SR
06 00 19  011E  41 29 0 0098            LA
06 00 19  0124  41 82 0 0028            LA
06 00 03  012A  58 FC 0 09CE            L
04 00 00  0130  05 1F                   BALR
04 00 00  0134  0008                    ****
06 00 00  0138  98 EF 0 9008            LM
```

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

****

ARBRE ABSTRAIT DECORE **A6**

****

*C87D3C	VAR	15	*C87D4C PAS DE CLIST	6	1	255					
*C87D4C	IDFDEF	15	*C87D5C *NIL PAS DE CLIST		4	60					
*C87D5C	OP	21	*C87D98 *087D88 *NIL 171 1 1500 CLIST VIDE		6 0	6 5	1 255	1548 0	55	1	1548
*C87D8E	DENOT	21	*NIL CLIST VIDE	10	10	26					
*C87D98	BEGOMES	18	*C87E00 *087D84 *087D34 DEREF 6			261	6	256-	4	255	
*087D84	CALL	19	*C87DF0 *087DD0 *NIL CLIST VIDE			6	6	1	0	5	0 255
*C87DDC	COMMAND	0	*C87DE0 *NIL PAS DE CLIST		315	1					
*C87DEC	DENOT	19	*NIL CLIST VIDE	10	10	2					
*C87DFC	IDF	18	*NIL CLIST VIDE	315	4-	3	0				
*C87E0C	INDEX	17	*C87E3C *087E1C *NIL CLIST VIDE			261	2-	1	0	1	255 255
*087E1C	COMMAND	0	*C87E2C *NIL PAS DE CLIST		0	0					
*C87E2C	DENOT	17	*NIL CLIST VIDE	10	10	2					

```
GRENOBLE ALGOL 68.                                                              01 DECEMBRE 1975  a  16:172400  PAGE   9
                      CLIST VIDE

 *0E7E3C  IDF            16      *NIL          318         4-         4        48
                      CLIST VIDE
*******************************************************************************************************************
GEN DESCENTE DANS NOEUD  VAR
                                          04 00 16  00BE  000F                        ****
GEN DESCENTE DANS NOEUD  OP
                                          04 00 16  00C2  0015                        ****
GEN DESCENTE DANS NOEUD  BECCMES
                                          04 00 16  00C6  0012                        ****
GEN DESCENTE DANS NOEUD  INDEX
                                          04 00 16  00CA  0011                        ****
GEN DESCENTE DANS NOEUD  DENCT
GEN REMONTER DANS NOEUD  COMMAIND
GEN DESCENTE DANS NOEUD  IDF
                                          04 00 16  00CE  0010                        ****
GEN REMONTER DANS NOEUD  INDEX
                                          02 02 11  00D2                              TAG    02,11
                                          12 19 18  00D4  D5 03 0 0000 9 0064         CLC
                                          06 00 00  00E0  45 8D 0 02CA                BAL
                                          02 00 12  00E6                              TAG    00,12
                                          06 00 19  00E8  58 39 0 0060                L
                                          06 00 21  00EE  58 1B 0 0000                L
                                          02 02 11  00F4                              TAG    02,11
                                          06 00 19  00F6  59 19 0 0074                C
                                          06 00 00  00FC  45 8C 0 0204                BAL
                                          06 00 19  0102  59 19 0 0078                C
                                          06 00 00  0108  45 8D 0 02DE                BAL
                                          02 00 12  010E                              TAG    00,12
                                          06 00 19  0110  5C 09 0 007C                M
                                          04 00 00  0116  1A 31                       AR
GEN REMONTER DANS NOEUD  BECCMES
GEN DESCENTE DANS NOEUD  CALL
                                          04 00 16  011A  0013                        ****
                                          06 00 19  011E  50 39 0 0098                ST
GEN DESCENTE DANS NOEUD  IDF
                                          04 00 16  0124  0012                        ****
GEN REMONTER DANS NOEUD  CALL
GEN DESCENTE DANS NOEUD  DENCT
                                          04 00 16  0128  0013                        ****
GEN REMONTER DANS NOEUD  COMMAIND
                                          06 00 00  012C  58 19 0 0018                L
                                          06 00 21  0132  58 6B 0 0000                L
                                          06 00 19  0138  50 61 0 0028                ST
                                          06 00 19  013E  98 EF 9 0030                LM
                                          04 00 00  0146  05 1F                       BALR
                                          06 00 00  014A  98 EF 0 9008                LM
GEN REMONTER DANS NOEUD  BECCMES
                                          06 00 19  0150  58 79 0 0098                L
                                          06 00 00  0156  60 27 0 0000                STD
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  DENCT
                                          04 00 16  015C  0015                        ****
GEN REMONTER DANS NOEUD  OP
                                          06 00 21  0160  58 6B 0 0014                L
                                          04 00 00  0166  10 76                       LPR
```

```
                                        06 00 19  016A  54 6C 0 0164        N
                                        06 00 19  0170  90 67 D 0118        STM
                                        06 00 19  0178  96 4E D 0118        DI
                                        04 00 00  0180  28 00               SDR
                                        06 00 19  0184  6A 6C 0 0118        AD
                                        04 00 00  018A  20 20               DDR
GEN REMONTER DANS NOEUD  IDFDEF

                                        06 00 19  018E  60 29 0 0050        STD
GEN REMONTER DANS NOEUD  DCHA
```

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

****

ARBRE ABSTRAIT DECORE **A7**

****

*C87A4C DO 32 *C87E3C *NIL *NIL *C87CB4 *C87A6C 7 0 8 13 23
24 0 0 25 255
PAS DE CLIST

*C87A6C DOWN 0 *C87A7C 7 4 0 255
PAS DE CLIST

*C87A7C OP 49 *C87CC4 *C87AA8 *C128A0 261 4 1 7776 164 1 7776
265 1 7680 0 5 255 0
METTRE 4

*C87AA8 BECOMES 50 *C87CB4 *C87AC4 *C879F0 261 6 256- 4 255
DEREF 6

*C87AC4 IFTE 51 *C87B5C *C87B00 *C87AF0 *C30B6E 1 256- 0 6 5 23
21 19 54 59 255
PAS DE CLIST

*C87AFC IDF 59 *C879F8 261 0 4 24
DEREF 6

*C87BCC DOWN 0 *C87B10 7 4 114 255
PAS DE CLIST

*C87B1C OP 58 *C87B4C *C87B3C *NIL 6 6 1 1548 55 1 1548
171 1 1500 0 5 255 0
CLIST VIDE

*C87B3C DENOT 58 *NIL 10 10 26
CLIST VIDE

*C87B4C IDF 58 *C87A00 261 0 4 24
DEREF 6

*C87B5C OP 53 *C87BC4 *C87B88 *NIL 13 13 1 60 109 1 60
155 0 0 0 3 255 0
CLIST VIDE

*087B8E	OP	53	*C87BC4 157 CLIST VIDE	*087BB4 0	*NIL 0	13 0	13 4	1 255	192 0	19	1	192
*087BB4	DENOT	53	*NIL CLIST VIDE	10	10	14						
*087BC4	IDF	53	*C87A08 DEREF	133 10	44	4	0					
*087BD4	OP	51	*087C78 165 CLIST VIDE	*087C00 1	*NIL 552	13 256-	13 4	1 255	912 0	19	1	912
*087C0C	OP	51	*C87C2C 73 CLIST VIDE	*NIL 1	*NIL 792	6 256-	6 4	1 255	1104 0	260	1	1104
*087C2C	INDEX	52	*C87C68 DEREF	*087C48 6	*087A10	261	6	1	256-	4	255	255
*087C48	COMMAND	C	*C87C58 PAS DE CLIST	*NIL	0	0						
*087C58	IDF	52	*C87A18 DEREF	133 10	10	4	0					
*087C6E	IDF	51	*NIL CLIST VIDE	318	4-	4	48					
*087C7E	OP	51	*087CA4 73 CLIST VIDE	*NIL 1	*NIL 792	6 256-	6 4	1 255	1104 0	260	1	1104
*087CA4	IDF	51	*C87A20 DEREF	261 6	15	4	24					
*087CB4	IDF	50	*NIL CLIST VIDE	261	2-	4	36					
*087CC4	IDF	49	*NIL CLIST VIDE	261	274	4	60					
*087CD4	OP	31	*C87D10	*087C00	*NIL	13	13	1	168	22	1	168

			157 CLIST VIDE	0	0	0	3	255	0			
*087DCC	IDF	31	*NIL CLIST VIDE	10	0	3	24					
*087D1C	BECOMES	26	*C87E2C DEREF	*087D2C 10	*087A28	133	10	256-	4	255		
*087D2C	IFTE	27	*C87DB4 12 PAS DE CLIST	*087D68 10	*087D58 28	*030B6E 29	1 255	256-	0	10	5	11
*087D58	DENOT	29	*NIL CLIST VIDE	10	10	2						
*087D68	OP	28	*C87DA4 159 CLIST VIDE	*087D94 0	*NIL 0	10 0	10 5	1 255	312 0	163	1	312
*087D94	DENOT	28	*NIL CLIST VIDE	10	10	2						
*087DA4	IDF	28	*C87A30 DEREF	133 10	14	4	12					
*087DB4	OP	27	*087DF0 165 CLIST VIDE	*087DE0 1	*NIL 552	13 0	13 3	1 255	912 0	19	1	912
*087DEC	IDF	27	*NIL CLIST VIDE	6	10	3	12					
*087DFC	OP	27	*087E1C 73 CLIST VIDE	*NIL 1	*NIL 792	6 256-	6 4	1 255	1104 0	260	1	1104
*087E1C	IDF	27	*C87A38 DEREF	261 6	15	4	36					
*087E2C	IDF	26	*NIL CLIST VIDE	133	2-	4	12					
*087E3C	DENOT	24	*NIL CLIST VIDE	10	10	26						

```
GEN DESCENTE DANS NOEUD  DO
                                 04 00 16  0194  0020                    ****
                                 06 00 21  0196  58 6B 0 0000            L
                                 06 00 19  019E  50 69 0 0098            ST
GEN DESCENTE DANS NOEUD  DENCT
                                 04 00 16  01A4  0018                    ****
GEN REMONTER DANS NOEUD  DO
                                 06 00 21  01A8  58 6B 0 0014            L
                                 06 00 19  01AE  50 69 0 0080            ST
                                 04 00 02  01B4                 CL.2     EQU   *
GEN DESCENTE DANS NOEUD  OP
                                 04 00 16  01B8  001F                    ****
GEN DESCENTE DANS NOEUD  BECOMES
                                 04 00 16  01BC  001A                    ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  BECOMES
GEN DESCENTE DANS NOEUD  IFTE
                                 04 00 16  01C0  0018                    ****
GEN DESCENTE DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP
                                 06 00 19  01C4  68 09 0 0048            LD
                                 04 00 00  01CA  20 00                   LPDR
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP
                                 06 00 19  01CE  69 09 0 0028            CD
GEN REMONTER DANS NOEUD  IFTE
                                 06 00 03  01D4  47 BC 0 0006            BC
GEN DESCENTE DANS NOEUD  OP
                                 04 00 16  01DA  001C                    ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  DENCT
GEN REMONTER DANS NOEUD  OP
                                 06 00 19  01DE  58 19 0 005C            L
                                 06 00 21  01E4  5A 1B 0 0000            A
GEN REMONTER DANS NOEUD  IFTE
                                 04 00 00  01EA  18 21                   LR
                                 06 00 03  01EE  47 F0 0 0005            BC
                                 04 00 02  01F4                 CL.6     EQU   *
GEN DESCENTE DANS NOEUD  DENCT
                                 04 00 16  01F8  001E                    ****
GEN REMONTER DANS NOEUD  IFTE
                                 06 00 21  01FC  58 2B 0 0000            L
                                 04 00 02  0202                 CL.5     EQU   *
                                 04 00 15  0206  0005                    ****
GEN REMONTER DANS NOEUD  BECOMES
                                 06 00 19  020A  50 29 0 005C            ST
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  IDF
                                 04 00 16  0210  001F                    ****
GEN REMONTER DANS NOEUD  OP
                                 06 00 19  0214  59 29 0 0038            C
GEN REMONTER DANS NOEUD  DO
                                 06 00 03  021A  47 2C 0 0004            BC
GEN DESCENTE DANS NOEUD  DOWN
```

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

ARBRE ABSTRAIT DECORE **A8**

```
*087934  BECOMES      32   *C8799C  *087950  *0128A0    261      4      0      5      255
                           NEUTRE       4

*08795C  CALL         33   *C8798C  *08796C  *NIL         6      6      1      0      5      0      255
                           CLIST VIDE

*08796C  COMMAND       0   *C8797C  *NIL       315       1
                           PAS DE CLIST

*08797C  IDF          33   *NIL       10       10       7      0
                           CLIST VIDE

*08798C  IDF          32   *NIL      315       4-       3      0
                           CLIST VIDE

*C8799C  IDF          32   *NIL      261       2-       4     24
                           CLIST VIDE
```

```
GEN DESCENTE DANS NOEUD  BECOMES

GEN DESCENTE DANS NOEUD  IDF                04 00 16  0220  0020                ****
GEN REMONTER DANS NOEUD  BECOMES
GEN DESCENTE DANS NOEUD  CALL

GEN DESCENTE DANS NOEUD  IDF                04 00 16  0224  0021                ****

GEN REMONTER DANS NOEUD  CALL               04 00 16  0228  0020                ****
GEN DESCENTE DANS NOEUD  IDF

GEN REMONTER DANS NOEUD  COMMAND            04 00 16  022C  0021                ****

                                            06 00 00  0230  58 19 0 0018         L
                                            06 00 19  0236  58 69 0 0080         L
                                            06 00 19  023C  50 61 0 0028         ST
                                            08 00 19  0242  98 EF 9 0030         LM
                                            04 00 00  024A  05 1F                BALR
                                            06 00 00  024E  98 EF 0 9008         LM
GEN REMONTER DANS NOEUD  BECOMES

GEN REMONTER DANS NOEUD  DOWN               06 00 19  0254  60 25 0 0040         STD
```

```
GRENOBLE ALGOL 68.                                                      01 DECEMBRE 1975  a  16:172400  PAGE  17

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION
*****************************************************************************************************************

         ARBRE ABSTRAIT DECORE   A9

*****************************************************************************************************************

*0E78EE  DO          38     *NIL      *NIL      *08799C  *NIL     *087914   14      0       0       15      36
                              0         0        37       0        255
                           PAS DE CLIST

*0E7914  DOWN        C      *C87924    14        5        0        255
                           PAS DE CLIST

*0E7924  BECOMES     47     *087950   *087940   *0128A0   261      4        0       5       255
                           NELIRE       4

*0E794C  IDF         47     *C878C8    261       6        4        36
                           DEREF        6

*08795C  INDEX       46     *C8798C   *08796C   *NIL      261      2-       1       0       1       255     255

                           CLIST VIDE

*0E796C  COMMAIND    C      *C8797C   *NIL       0        0
                           PAS DE CLIST

*0E797C  IDF         46     *NIL       10        10       14       0
                           CLIST VIDE

*0E798C  IDF         45     *NIL       318       4-       4        48
                           CLIST VIDE

*0E799C  IDF         37     *C878E0    133       10       4        0
                           DEREF        10
*****************************************************************************************************************
GEN DESCENTE DANS NOEUD  DC
                                        04 00 16  025A  0026                      ****
                                        06 00 21  025E  58 68 0 0000              L
                                        06 00 19  0264  50 69 0 00A4              ST

GEN DESCENTE DANS NOEUD  IDF
                                        04 00 16  026A  0025                      ****
GEN REMONTER DANS NOEUD  DC
                                        06 00 19  026E  58 69 0 0058              L
                                        06 00 19  0274  50 69 0 00A8              ST
                                        06 00 21  027A  58 68 0 0000              L
```

```
06 00 19  0280  50 69 0 0088          ST
06 00 19  0286  58 29 0 0088          L
06 00 19  028C  55 29 0 00A8          C
06 00 03  0292  47 20 0 0009          BC
04 00 02  0298                LL.7    EWU   *
06 00 19  029C  50 29 0 0088          ST
```

GEN DESCENTE DANS NOEUD DCWA

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

ARBRE ABSTRAIT DECORE **A10**

*C8778C	BECOMES	38	*C8788C NEUTRE	*C8779C 4	*C128A0	261	4	0	5	255		
*C8779C	OP	44	*C8778C 171 CLIST VIDE	*C8778C 1	*NIL 1500	6 0	6 5	1 255	1548 0	55	1	1548
*C8778C	DENOT	44	*NIL CLIST VIDE	10	10	26						
*C8778C	OP	41	*C87850 167 CLIST VIDE	*C87804 1	*NIL 672	6 256-	6 4	1 255	1032 0	103	1	1032
*C8780C	INDEX	42	*C87840 DEREF	*C87820 6	*C87768	261	6	1	256-	4	255	255
*C8782C	COMMAND	0	*C87830 PAS DE CLIST	*NIL	0	0						
*C8783C	IDF	42	*NIL CLIST VIDE	10	10	14	0					
*C8784C	IDF	41	*NIL CLIST VIDE	318	4-	4	48					
*C8785C	BECOMES	40	*C8787C DEREF	*C8786C 6	*C87770	261	6	256-	4	255		
*C8786C	IDF	40	*C87778 DEREF	261 6	0	4	24					
*C8787C	IDF	40	*NIL CLIST VIDE	261	2-	4	36					
*C8788C	IDF	38	*NIL	261	2-	4	24					

CLIST VIDE

```
GEN DESCENTE DANS NOEUD  BECOMES
                                  04 00 16 02A2  0026                    ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  BECOMES
GEN DESCENTE DANS NOEUD  CP
                                  04 00 16 02A6  002C                    ****
GEN DESCENTE DANS NOEUD  OP
                                  04 00 16 02AA  0029                    ****
GEN DESCENTE DANS NOEUD  BECOMES
                                  04 00 16 02AE  0028                    ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  BECOMES
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  BECOMES
                                  12 19 19 02B2  D2 07 9 0048 9 0040      MVC
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  INDEX
                                  04 00 16 02BE  002A                    ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  COMPAINL
GEN DESCENTE DANS NOEUD  IDF
                                  04 00 16 02C2  0029                    ****
GEN REMONTER DANS NOEUD  INDEX
                                  02 02 11 02C6                          TAG   02,11
                                  12 19 18 02C8  D5 03 0 0000 9 0064      CLC
                                  06 00 00 02D4  45 8D 0 02CA            BAL
                                  02 00 12 02DA                          TAG   00,12
                                  06 00 19 02DC  58 39 0 0060            L
                                  06 00 19 02E2  58 19 0 0088            L
                                  02 02 11 02E8                          TAG   02,11
                                  06 00 19 02EA  59 19 0 0074            C
                                  06 00 00 02F0  45 8D 0 02D4            BAL
                                  06 00 19 02F6  59 19 0 0078            C
                                  06 00 00 02FC  45 8D 0 02DE            BAL
                                  02 00 12 0302                          TAG   00,12
                                  06 00 19 0304  5C 09 0 007C            M
                                  04 00 00 030A  1A 31                   AR
GEN REMONTER DANS NOEUD  OP
                                  06 00 19 030E  68 09 0 0040            LD
                                  06 00 19 0314  6A 03 0 0000            AD
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  DENOT
                                  04 00 16 031A  002C                    ****
GEN REMONTER DANS NOEUD  CP
                                  06 00 21 031E  58 6B 0 0014            L
                                  04 00 00 0324  10 76                   LPR
                                  06 00 19 0328  54 6D 0 0164            N
                                  08 00 19 032E  90 67 D 0118            STM
                                  08 00 19 0336  98 4E D 0118            LI
                                  04 00 00 033E  2B 22                   SDR
                                  06 00 19 0342  6A 2D 0 0118            AD
                                  04 00 00 0348  2D 02                   DDR
GEN REMONTER DANS NOEUD  BECOMES
                                  06 00 19 034C  60 09 0 0040            STD
GEN REMONTER DANS NOEUD  DOWN
GEN DESCENTE DANS NOEUD  BECOMES
```

```
                                                 04 00 16  0352  002F                          ****

GEN DESCENTE DANS NOEUD  INDEX

                                                 04 00 16  0356  002E                          ****

GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  COMMAINC
GEN DESCENTE DANS NOEUD  IDF

                                                 04 00 16  035A  002D                          ****

GEN REMONTER DANS NOEUD  INDEX

                                                 02 02 11  035E                              TAG   02,11
                                                 12 19 18  0360  D5 03 0 0000 9 0064         CLC
                                                 00 00 00  0366  45 8D 0 02CA                BAL
                                                 02 00 12  0372                              TAG   00,12
                                                 06 00 19  0374  58 29 0 0060                L
                                                 06 00 19  037A  58 59 0 0088                L
                                                 02 02 11  0380                              TAG   02,11
                                                 06 00 19  0382  59 59 0 0074                C
                                                 00 00 00  0388  45 8D 0 02D4                BAL
                                                 06 00 19  038E  59 59 0 0078                C
                                                 06 00 00  0394  45 8D 0 02DE                BAL
                                                 02 00 12  039A                              TAG   00,12
                                                 06 00 19  039C  5C 49 0 007C                M
                                                 04 00 00  03A2  1A 25                       AR

GEN REMONTER DANS NOEUD  BECOMES
GEN DESCENTE DANS NOEUD  IDF

                                                 04 00 16  03A6  002F                          ****

GEN REMONTER DANS NOEUD  BECOMES

                                                 12 19 19  03AA  D2 07 2 0000 9 0048         MVC

GEN REMONTER DANS NOEUD  DOWN
GEN REMONTER DANS NOEUD  DC

                                                 06 00 19  03B6  58 29 0 0088                L
                                                 06 00 19  03BC  98 01 9 00A4                LM
                                                 06 00 03  03C4  87 2C 0 0007                BXLE
                                                 04 00 02  03CC                      CC.9    EQU   *
                                                 04 00 13  03D0  0005                        ****

GEN REMONTER DANS NOEUD  DOWN
GEN DESCENTE DANS NOEUD  OP

                                                 04 00 16  03D4  0031                          ****

GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  BECOMES

                                                 04 00 16  03D8  0032                          ****

GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  BECOMES
GEN DESCENTE DANS NOEUD  IFTE

                                                 04 00 16  03DC  0033                          ****

GEN DESCENTE DANS NOEUD  OP

                                                 04 00 16  03E0  0035                          ****

GEN DESCENTE DANS NOEUD  OP

                                                 04 00 16  03E4  0033                          ****

GEN DESCENTE DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP

                                                 06 00 19  03E8  68 09 0 0040                LD
                                                 04 00 00  03EE  20 00                       LPDR

GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  INDEX

                                                 04 00 16  03F2  0034                          ****
```

```
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  COMMAND
GEN DESCENTE DANS NOEUD  IDF

GEN REMONTER DANS NOEUD  INDEX

                                  04 00 16  03F6  0033                    ****

                                  02 02 11  03FA                          TAG   02,11
                                  12 19 18  03FC  D5 03 0 0000 9 0064      CLC
                                  06 00 00  0408  45 8C 0 02CA            BAL
                                  02 00 12  040E                          TAG   00,12
                                  06 00 19  0410  58 39 0 0060            L
                                  06 00 19  0416  58 19 0 0058            L
                                  02 02 11  041C                          TAG   02,11
                                  06 00 19  041E  59 19 0 0074            C
                                  06 00 00  0424  45 8C 0 02D4            BAL
                                  06 00 19  042A  59 19 0 0078            C
                                  06 00 00  0430  45 8C 0 02DE            BAL
                                  02 00 12  0436                          TAG   00,12
                                  06 00 19  0438  5C 09 0 007C            M
                                  04 00 00  043E  1A 31                   AR

GEN REMONTER DANS NOEUD  OP       06 00 19  0442  68 23 0 0000            LD
                                  04 00 00  0448  20 22                   LPDR

GEN REMONTER DANS NOEUD  OP       04 00 00  044C  29 02                   CDR

GEN REMONTER DANS NOEUD  OP       06 00 19  0450  41 50 0 0000            LA
                                  06 00 03  0456  47 80 0 0005            BC
                                  06 00 19  045C  41 56 0 0001            LA
                                  04 00 02  0462                CL.5      EQU   *
                                  04 00 13  0466  0005                    ****

GEN DESCENTE DANS NOEUD  OP       04 00 16  046A  0035                    ****

GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  DENOT
GEN REMONTER DANS NOEUD  OP

                                  06 00 19  046E  58 29 0 0058            L
GEN REMONTER DANS NOEUD  OP       06 00 21  0474  59 28 0 0010            C

                                  06 00 19  047A  41 40 0 0000            LA
                                  06 00 03  0480  47 80 0 0005            BC
                                  06 00 19  0486  41 46 0 0001            LA
                                  04 00 02  048C                CL.5      EQU   *
                                  04 00 13  0490  0005                    ****
GEN REMONTER DANS NOEUD  IFTE     04 00 00  0494  14 54                   NR

                                  04 00 00  0498  12 55                   LTR
GEN DESCENTE DANS NOEUD  OCWN     06 00 03  049C  47 8C 0 0006            BC
```

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION

****

ARBRE ABSTRAIT DECORE **A11**

****

```
*C878FE  OP          54   *C87934   *087924   *0128A0    133        4        1      7536      164        1      7536
                           259         1        7592        0        5      255        0
                         NEUTRE        4

*C87924  DENOT       54    *NIL       10         10         2
                         CLIST VIDE

*C87934  IDF         54    *NIL      133        274         4        0
                         CLIST VIDE
```

****

```
GEN DESCENTE DANS NOEUD  OP
                                                 04 00 16  04A2  003C                    ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  DENOT
GEN REMONTER DANS NOEUD  OP
                                                 06 00 19  04A6  58 19 0 0058           L
                                                 06 00 21  04AC  5A 1B 0 0000           A
                                                 06 00 19  04B2  50 19 0 0058           ST
GEN REMONTER DANS NOEUD  DOWN
```

```
GRENOBLE ALGOL 68.                                                                     01 DECEMBRE 1975  à  16:172400  PAGE  24

*CONSTRUCTION DE L'ARBRE PRIMITIF
*PHASE DE DECORATION
*******************************************************************************************************************************

         ARBRE ABSTRAIT DECCRE  A12

*******************************************************************************************************************************

*CE78CC  BECOMES       57     *CE78F8    *0878E8    *0128A0     261        4        0        5       255
                              NEUTRE        4

*0878E8  IDF           57     *C878EC     261          0          4       24
                              DEREF         6

*CE78F8  INDEX         56     *087934    *087914    *NIL        261        2-       1        0        1       255       255

                              CLIST VIDE

*CE7914  COMMAINO      0      *087924    *NIL         0          0
                              PAS DE CLIST

*CE7924  IDF           56     *C878C4     133         10          4        0
                              DEREF        10

*CE7934  IDF           55     *NIL        318          4-         4       48
                              CLIST VIDE
*******************************************************************************************************************************
GEN DESCENTE DANS NOEUD  BECOMES
                                                   04 00 16  0488  0039                                ****
GEN DESCENTE DANS NOEUD  INDEX
                                                   04 00 16  04BC  0038                                ****
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  COMMAINO
GEN DESCENTE DANS NOEUD  IDF
                                                   04 00 16  04C0  0037                                ****
GEN REMONTER DANS NOEUD  INDEX
                                                   02 02 11  04C4                                      TAG   02,11
                                                   12 19 18  04C6  D5 03 0 0000 9 0064                 CLC
                                                   06 00 00  04D2  45 8C 0 02CA                        BAL
                                                   02 00 12  04D8                                      TAG   00,12
                                                   06 00 19  04CA  58 39 0 0060                        L
                                                   06 00 19  04E0  58 19 0 0058                        L
                                                   02 02 11  04E6                                      TAG   02,11
                                                   06 00 19  04E8  59 19 0 0074                        C
                                                   06 00 00  04EE  45 8C 0 0204                        BAL
                                                   06 00 19  04F4  59 19 0 0078                        C
                                                   06 00 00  04FA  45 8C 0 02DE                        BAL
                                                   02 00 12  0500                                      TAG   00,12
                                                   06 00 19  0502  5C 09 0 007C                        M
                                                   04 00 00  0508  1A 31                               AR
```

```
GRENOBLE ALGOL 68.                                                    01 DECEMBRE 1975 @ 16:172400  PAGE  25
GEN REMONTER DANS NOEUD   BECCMES
GEN DESCENTE DANS NOEUD   IDF
                                        04 00 16  050C  0039                           ****
GEN REMONTER DANS NOEUD   BECCMES
                                        12 19 19  0510  D2 07 3 0000 9 0040            MVC
GEN REMONTER DANS NOEUD   DOWN
GEN DESCENTE DANS NOEUD   OP
                                        04 00 16  051C  003A                           ****
GEN DESCENTE DANS NOEUD   IDF
GEN REMONTER DANS NOEUD   OP
GEN DESCENTE DANS NOEUD   DENOT
GEN REMONTER DANS NOEUD   OP
                                        06 00 21  0520  58 6B 0 0014                   L
                                        04 00 00  0526  10 7E                          LPR
                                        06 00 19  052A  54 6D 0 0164                   N
                                        08 00 19  0530  90 67 D 0118                   STM
                                        08 00 19  0538  96 4E D 0118                   OI
                                        04 00 00  0540  2B 44                          SDR
                                        06 00 19  0544  6A 4D 0 0118                   AD
                                        06 00 19  054A  68 69 0 0040                   LD
                                        04 00 00  0550  2D 64                          DDR
GEN REMONTER DANS NOEUD   DOWN
GEN REMONTER DANS NOEUD   IFTE
                                        04 00 00  0554  28 26                          LDR
                                        06 00 03  0558  47 F0 0 0005                   BC
                                        04 00 02  055E                        CL.6     EQU   *
GEN DESCENTE DANS NOEUD   IDF
                                        04 00 16  0562  003B                           ****
GEN REMONTER DANS NOEUD   IFTE
                                        06 00 19  0566  68 29 0 0040                   LD
                                        04 00 02  056C                        CL.5     EQU   *
                                        04 00 13  0570  0005                           ****
GEN REMONTER DANS NOEUD   BECCMES
                                        06 00 19  0574  60 29 0 0048                   STD
GEN REMONTER DANS NOEUD   OP
                                        06 00 19  057A  68 09 0 0050                   LD
                                        04 00 00  0580  2A 02                          ADR
                                        06 00 19  0584  60 C9 0 0050                   STD
GEN REMONTER DANS NOEUD   DOWN
GEN REMONTER DANS NOEUD   DC
                                        06 00 19  058A  58 29 0 0080                   L
                                        06 00 19  0590  5A 29 0 0098                   A
                                        06 00 19  0596  50 29 0 0080                   ST
                                        06 00 03  059C  47 F0 0 0002                   BC
                                        04 00 02  05A2                        CL.4     EQU   *
                                        04 00 13  05A6  0000                           ****
GEN REMONTER DANS NOEUD   DOWN
GEN DESCENTE DANS NOEUD   IDF
                                        04 00 16  05AA  003F                           ****
GEN REMONTER DANS NOEUD   DOWN
GEN REMONTER DANS NOEUD   BEGIN
                                        06 00 19  05AE  68 29 0 0050                   LD
GEN REMONTER DANS NOEUD   ROUTINE
                                        04 00 08  05B4  00C8                           ****
GEN REMONTER DANS NOEUD   IDFDEF
                                        06 00 20  0008  50 CC 0 0028                   ST
                                        06 00 03  000E  58 6C 0 0A28                   L
                                        06 00 20  0014  50 6C 0 002C                   ST
```

```
GEN REMONTER DANS NOEUD  DOWN
GEN DESCENTE DANS NOEUD  ENTSCRT

GEN DESCENTE DANS NOEUD  IDF

GEN REMONTER DANS NOEUD  ENTSCRT
GEN DESCENTE DANS NOEUD  CALL

GEN DESCENTE DANS NOEUD  IDF

GEN REMONTER DANS NOEUD  CALL
GEN DESCENTE DANS NOEUD  RCUTINE

GEN DESCENTE DANS NOEUD  IFTE

GEN DESCENTE DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP

GEN REMONTER DANS NOEUD  IFTE

GEN DESCENTE DANS NOEUD  OP

GEN DESCENTE DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  DENCT
GEN REMONTER DANS NOEUD  OP

GEN REMONTER DANS NOEUD  OP
GEN DESCENTE DANS NOEUD  IDF
GEN REMONTER DANS NOEUD  OP

GEN REMONTER DANS NOEUD  IFTE
```

```
04 00 16  001A  0042                              ****

04 00 16  001E  0041                              ****

04 00 16  0022  0043                              ****

04 00 16  0026  0042                              ****

04 00 16  002A  0044                              ****
04 00 02  0000                          GL.2602   EQU   *
04 00 07  0004  0044                              ****
06 00 19  0008  41 79 0 0030                      LA
00 00 01  000E  5079001400E2037000E000            TAGC  21
                58F011AA4120D18C4140
                58F0

04 00 16  0028  0046                              ****

06 00 19  0026  58 19 0 0028                      L
06 00 19  0032  54 1D 0 0248                      N

04 00 00  0038  12 11                             LTR
06 00 03  003C  47 80 0 0001                      BC

04 00 16  0042  0047                              ****

06 00 21  0046  58 3B C 0000                      L
04 00 00  004C  13 33                             LCR

04 00 00  0050  18 63                             LR
04 00 00  0054  10 76                             LPR
06 00 19  0058  54 6D 0 0164                      N
08 00 19  005E  90 67 D 0118                      STM
08 00 19  0066  96 4E D 0118                      OI
04 00 00  006E  2B 00                             SDR
06 00 19  0072  6A 00 0 0118                      AD
06 00 19  0078  58 69 0 0028                      L
04 00 00  007E  10 76                             LPR
06 00 19  0082  54 6D 0 0164                      N
08 00 19  0088  90 67 D 0118                      STM
08 00 19  0090  96 4E D 0118                      OI
04 00 00  0098  2B 22                             SDR
06 00 19  009C  6A 2E 0 0118                      AD
04 00 00  00A2  2B 02                             UDR

04 00 00  00A6  28 20                             LDR
06 00 03  00AA  47 FC C 0000                      BC
04 00 02  0080                          GL.1      EQU   *
```

```
GRENOBLE ALGOL 68.                                                                    01 DECEMBRE 1975  @  16:172400  PAGE  27

GEN DESCENTE DANS NOEUD   OP
                                        04 00 16  0084  0048                                     ****
GEN DESCENTE DANS NOEUD   DENCT
GEN REMONTER DANS NOEUD   OP
GEN DESCENTE DANS NOEUD   IDF
GEN REMONTER DANS NOEUD   OP

                                        06 00 21  0088  58 68 0 0000                             L
                                        04 00 00  00BE  10 7E                                    LPR
                                        06 00 19  00C2  54 6C 0 0164                             N
                                        08 00 19  00C8  90 67 D 0118                             STM
                                        08 00 19  00D0  96 4E D 0118                             OI
                                        04 00 00  00D8  2B 44                                    SDR
                                        06 00 19  00DC  6A 4C 0 0118                             AD
                                        06 00 19  00E2  58 69 0 0028                             L
                                        04 00 00  00E8  10 76                                    LPR
                                        06 00 19  00EC  54 6C 0 0164                             N
                                        08 00 19  00F2  90 67 D 0118                             STM
                                        08 00 19  00FA  96 4E D 0118                             OI
                                        04 00 00  0102  2B 66                                    SDR
                                        06 00 19  0106  6A 6D 0 0118                             AD
                                        04 00 00  010C  2C 46                                    DDR
GEN REMONTER DANS NOEUD   IFTE
                                        04 00 00  0110  28 24                                    LDR
                                        04 00 02  0114                         CL.0              EQU    *
                                        04 00 13  0118  0000                                     ****
GEN REMONTER DANS NOEUD   ROUTINE
                                        04 00 08  011C  0038                                     ****
GEN REMONTER DANS NOEUD   COMPAIND
GEN DESCENTE DANS NOEUD   DENCT
                                        04 00 16  002E  004A                                     ****
GEN REMONTER DANS NOEUD   COMPAIND
GEN DESCENTE DANS NOEUD   DENCT
                                        04 00 16  0032  0048                                     ****
GEN REMONTER DANS NOEUD   COMPAINC
                                        06 00 00  0036  58 19 0 0018                             L
                                        06 00 19  003C  50 C1 0 0030                             ST
                                        06 00 03  0042  58 60 0 0A2A                             L
                                        06 00 19  0048  50 61 0 0034                             ST
                                        12 21 19  004E  D2 07 1 0028 B 0018                      MVC
                                        06 00 21  005A  58 6B 0 0014                             L
                                        06 00 19  0060  50 61 0 0038                             ST
                                        08 00 20  0066  98 EF C 0028                             LM
                                        04 00 00  006E  05 1F                                    BALR
                                        06 00 00  0072  98 EF 0 9008                             LM
GEN REMONTER DANS NOEUD   COMPAIND
                                        06 00 20  0078  60 2C 0 0030                             STD
                                        06 00 19  007E  41 30 0 0004                             LA
                                        06 00 20  0084  68 2C 0 0030                             LD
                                        06 00 03  008A  58 F0 0 09C9                             L
                                        04 00 00  0090  05 1F                                    BALR
                                        06 00 00  0094  98 EF 0 9008                             LM

GEN REMONTER DANS NOEUD   DOWN
GEN REMONTER DANS NOEUD   BEGIN
GEN REMONTER DANS NOEUD   ICASCUT
                                        02 00 09  009A                                           TAG    00,09
```

EDITION DE LA TABLE DES PARENTHESES

	BLOC NO 2	ADDTIM 1C29-	ADDTIP 1029-	ADDTID 11496	ADDTOP 1029-	NIVSTAT 1	IGG 2	TPSI 8	DEPLPSI 40
DEPL	---------TID--------	NUMBLKTID	INCTUTID	NUMIDTID	CLASSTID	DEPLTID	STOUET		
0		0	0	514	316	40	1		
	BLOC NO 3	1C29-	1029-	11412	1029-	2	3	24	40
DEPL	---------TID--------	NUMBLKTID	INCTUTID	NUMIDTID	CLASSTID	DEPLTID	STOUET		
0		0	0	304	315	48	1		
12		0	0	515	6	40	1		
24		0	0	516	16	56	1		
	BLOC NO 4	1C29-	1029-	11322	1029-	2	4	64	64
DEPL	---------TID--------	NUMBLKTID	INCTUTID	NUMIDTID	CLASSTID	DEPLTID	STOUET		
0		0	0	312	133	88	2		
12		0	0	313	133	92	2		
24		0	0	317	261	64	2		
36		0	0	318	261	72	2		
48		0	0	311	318	96	2		
60		0	0	319	261	80	2		
	BLOC NO 7	1C29-	1029-	11292	1029-	2	5	8	128
DEPL	---------TID--------	NUMBLKTID	INCTUTID	NUMIDTID	CLASSTID	DEPLTID	STOUET		
0		0	0	307	10	128	1		
	BLOC NO 14	1C29-	1029-	11262	1029-	2	6	6	136
DEPL	---------TID--------	NUMBLKTID	INCTUTID	NUMIDTID	CLASSTID	DEPLTID	STOUET		
0		0	0	309	16	136	1		
	BLOC NO 22	1C29-	1029-	11406	1029-	2	3	8	40
DEPL	---------TID--------	NUMBLKTID	INCTUTID	NUMIDTID	CLASSTID	DEPLTID	STOUET		
0		0	0	307	16	40	1		

EDITION DE LA TABLE DES CLASSES

		CLASSPTR	STATTAILLE	ALIGNEMENT	PROFONDEUR			
CLASSE NO	315	4416	8	2	256			
-----TYPEPROC4------		CODEPROC4	NOCLASSP4	SAMSTAP4	NBPARAM4	RESULT	PARAM4	DEPLPARAM4
		45	315	1442	1	6	10	0
CLASSE NO	316	4430	8	2	256			
-----TYPEPROC4------		CODEPROC4	NOCLASSP4	SAMSTAP4	NBPARAM4	RESULT	PARAM4	DEPLPARAM4
		45	316	4086	3	6	315	8
							6	0
							10	16
CLASSE NO	317	4452	32	2	0			
-----TYPEROW--------		CODEROW	NOCLASSROW	SAMSTAROW	SSDECROW	DIMENS		
		43	317	3864	6	1		
CLASSE NO	318	4462	32	2	0			
-----TYPEREF--------		CODEREF	NOCLASSREF	SAMSTAREF	SSDECREF			
		39	318	4264	317			

La trace de chargement

Quatre textes de routine ont été produits pendant la génération de code :

- routine associée au déclareur du tableau M : RT0002
- routine possédée par l'identificateur : RT0001
- routine associée au premier paramètre d'euler : RT0003
- routine associée au programme principal : RT0000

La trace comprend l'adresse de chargement, le déplacement de chaque instruction dans la zone tampon et l'instruction elle-même, sous forme hexadécimale et sous forme partiellement décodée.

Elle fournit enfin l'implantation de la table des constantes et de chaque texte de routine.

```
FILE = EULER2    STANDOUT P1:
                 CHARGEMENT ALGOL68    EULER2                                    01 DECEMBRE 1975  @  16:193400   PAGE    1

CHARGEMENT DU PROGRAMME "EULER2   " COMPILE LE    01 DECEMBRE 1975  @  16:172400

LA LISTE COMPLETE DES OPTIONS DE CHARGEMENT EST --

                                                    CODE
                                                    DUMP
                                                  NOTRACE
                                                  NOUNITE
                                                  NOVERIF
                                                  NOMSG
                                                  NOTEST
                                                    STACK=       3K
```

```
01A110  000000  47 FC F 0CE          BC    15,14(R0,R15)
01A114  000004  0CC0CCC0             DC    X'0CC0CCC0'
01A118  000008  0C0CCC48             DC    X'0C0CCC48'
01A11C  00000C  0000                 DC    X'0000'
01A11E  00000E  50 10 D 588          ST    R1,1480(R0,R13)
01A122  000012  90 EF 9 008          STM   R14,R15,8(R9)
01A126  000016  50 CC C 028          ST    R12,40(R12,R0)
01A12A  00001A  58 60 C 038          L     R6,56(R0,R12)
01A12E  00001E  50 6C C 02C          ST    R6,44(R12,R0)
01A132  000022  58 19 C 018          L     R1,24(R9,R0)
01A136  000026  50 C1 C 030          ST    R12,48(R1,R0)
01A13A  00002A  58 60 C 040          L     R6,64(R0,R12)
01A13E  00002E  50 61 C 034          ST    R6,52(R1,R0)
01A142  000032  D2 07 1 028  8 018   MVC   40(8,R1),24(R11)
01A148  000038  58 6B C 014          L     R6,20(R11,R0)
01A14C  00003C  50 61 C 038          ST    R6,56(R1,R0)
01A150  000040  98 EF C 028          LM    R14,R15,40(R12)
01A154  000044  05 1F                BALR  R1,R15
01A156  000046  98 EF 9 008          LM    R14,R15,8(R9)
01A15A  00004A  60 2C C 030          STD   F2,48(R12,R0)
01A15E  00004E  41 30 C 004          LA    R3,4(R0,R0)
01A162  000052  68 2C C 030          LD    F2,48(R12,R0)
01A166  000056  58 F0 D 61C          L     R15,1564(R0,R13)
01A16A  00005A  05 1F                BALR  R1,R15
01A16C  00005C  98 EF 9 008          LM    R14,R15,8(R9)
01A17C  000060  58 10 D 588          L     R1,1480(R0,R13)
01A174  000064  07 F1                BCR   R15,R1
```

```
01A178  000000   47 F0 F 00E          BC    15,14(R0,R15)
01A17C  000004   0CCCCCC0             DC    X'0CCCCCC0'
01A180  000008   0C0C0C38             DC    X'0C0C0C38'
01A184  00000C   0044                 DC    X'0044'
01A186  00000E   45 80 D 300          BAL   R8,768(R0,R13)
01A18A  000012   41 79 C 03C          LA    R7,48(R9,R0)
01A18E  000016   50 79 C 014          ST    R7,20(R9,R0)
01A192  00001A   D2 03 7 000  E 000   MVC   0(4,R7),0(R14)
01A198  000020   50 97 C 004          ST    R9,4(R7,R0)
01A19C  000024   58 E7 C 000          L     R14,0(R7,R0)
01A1A0  000028   90 EF 9 008          STM   R14,R15,8(R9)
01A1A4  00002C   58 19 C 028          L     R1,40(R9,R0)
01A1A8  000030   54 1D C 248          N     R1,584(R13,R0)
01A1AC  000034   12 11                LTR   R1,R1
01A1AE  000036   47 CC C 000          BC    0,0(R0,R0)
01A1B2  00003A   47 8F C 07E          BC    8,126(R15,R0)
01A1B6  00003E   58 3B C 000          L     R3,0(R11,R0)
01A1BA  000042   13 33                LCR   R3,R3
01A1BC  000044   18 63                LR    R6,R3
01A1BE  000046   1C 76                LPR   R7,R6
01A1C0  000048   54 6D C 164          N     R6,356(R13,R0)
01A1C4  00004C   90 67 D 118          STM   R6,R7,280(R13)
01A1C8  000050   96 4E D 118          CI    280(R13),X'4E'
01A1CC  000054   2B C0                SDR   F0,F0
01A1CE  000056   6A 0D C 118          AD    F0,280(R13,R0)
01A1D2  00005A   58 69 C 028          L     R6,40(R9,R0)
01A1D6  00005E   1C 76                LPR   R7,R6
01A1D8  000060   54 6D C 164          N     R6,356(R13,R0)
01A1DC  000064   90 67 D 118          STM   R6,R7,280(R13)
01A1E0  000068   96 4E D 118          CI    280(R13),X'4E'
01A1E4  00006C   2B 22                SDR   F2,F2
01A1E6  00006E   6A 2D C 118          AD    F2,280(R13,R0)
01A1EA  000072   2D 02                DDR   F0,F2
01A1EC  000074   28 20                LDR   F2,F0
01A1EE  000076   47 CC C 000          BC    0,0(R0,R0)
01A1F2  00007A   47 FF 0 0B2          BC    15,178(R15,R0)
01A1F6  00007E   58 6B C 000          L     R6,0(R11,R0)
01A1FA  000082   1C 76                LPR   R7,R6
01A1FC  000084   54 6D C 164          N     R6,356(R13,R0)
01A200  000088   90 67 D 118          STM   R6,R7,280(R13)
01A204  00008C   96 4E D 118          CI    280(R13),X'4E'
01A208  000090   2B 44                SDR   F4,F4
01A20A  000092   6A 4D C 118          AD    F4,280(R13,R0)
01A20E  000096   58 69 C 028          L     R6,40(R9,R0)
01A212  00009A   1C 76                LPR   R7,R6
01A214  00009C   54 6D C 164          N     R6,356(R13,R0)
01A218  0000A0   90 67 D 118          STM   R6,R7,280(R13)
01A21C  0000A4   96 4E D 118          CI    280(R13),X'4E'
01A220  0000A8   2B 66                SDR   F6,F6
01A222  0000AA   6A 6D C 118          AD    F6,280(R13,R0)
01A226  0000AE   2D 46                DDR   F4,F6
01A228  0000B0   28 24                LDR   F2,F4
01A22A  0000B2   58 9C 9 000          L     R9,0(R0,R9)
01A22E  0000B6   58 10 9 004          L     R1,4(R0,R9)
01A232  0000BA   07 F1                BCR   R15,R1
```

```
01A238  00000000  47 F0 F 0CE            BC    15,14(R0,R15)
01A23C  00000004  0C000000               DC    X'00000000'
01A240  00000008  0C0000C8               DC    X'000000C8'
01A244  0000000C  0002                   DC    X'0002'
01A246  0000000E  45 80 D 300            BAL   R8,768(R0,R13)
01A24A  00000012  41 79 C 090            LA    R7,144(R9,R0)
01A24E  00000016  50 79 C 014            ST    R7,20(R9,R0)
01A252  0000001A  D2 03 7 000  E 000     MVC   0(4,R7),0(R14)
01A258  00000020  50 97 C 004            ST    R9,4(R7,R0)
01A25C  00000024  58 E7 C 000            L     R14,0(R7,R0)
01A260  00000028  90 EF 9 008            STM   R14,R15,8(R9)
01A264  0000002C  58 6B C 000            L     R6,0(R11,R0)
01A268  00000030  50 69 C 058            ST    R6,88(R9,R0)
01A26C  00000034  D2 07 9 048  9 028     MVC   72(8,R9),40(R9)
01A272  0000003A  41 29 C 098            LA    R2,152(R9,R0)
01A276  0000003E  41 50 C 001            LA    R5,1(R0,R0)
01A27A  00000042  58 FC C 03C            L     R15,60(R0,R12)
01A27E  00000046  18 EC                  LR    R14,R12
01A280  00000048  05 1F                  BALR  R1,R15
01A282  0000004A  98 EF 9 0C8            LM    R14,R15,8(R9)
01A286  0000004E  41 30 C 001            LA    R3,1(R0,R0)
01A28A  00000052  58 F0 D 618            L     R15,1560(R0,R13)
01A28E  00000056  05 1F                  BALR  R1,R15
01A290  00000058  0004                   DC    X'0004'
01A292  0000005A  98 EF 9 0C8            LM    R14,R15,8(R9)
01A296  0000005E  18 53                  LR    R5,R3
01A298  00000060  41 39 C 060            LA    R3,96(R9,R0)
01A29C  00000064  1B 66                  SR    R6,R6
01A29E  00000066  1B 77                  SR    R7,R7
01A2A0  00000068  41 29 C 098            LA    R2,152(R9,R0)
01A2A4  0000006C  41 82 C 028            LA    R8,40(R2,R0)
01A2A8  00000070  58 F0 D 618            L     R15,1560(R0,R13)
01A2AC  00000074  05 1F                  BALR  R1,R15
01A2AE  00000076  0C08                   DC    X'0008'
01A2B0  00000078  98 EF 9 0C8            LM    R14,R15,8(R9)
01A2B4  0000007C  58 39 C 060            L     R3,96(R9,R0)
01A2B8  00000080  58 1B C 000            L     R1,0(R11,R0)
01A2BC  00000084  5C 09 C 07C            M     R0,124(R9,R0)
01A2C0  00000088  1A 31                  AR    R3,R1
01A2C2  0000008A  50 39 C 098            ST    R3,152(R9,R0)
01A2C6  0000008E  58 19 C 018            L     R1,24(R9,R0)
01A2CA  00000092  58 6B C 000            L     R6,0(R11,R0)
01A2CE  00000096  50 61 C 028            ST    R6,40(R1,R0)
01A2D2  0000009A  98 EF 9 030            LM    R14,R15,48(R9)
01A2D6  0000009E  05 1F                  BALR  R1,R15
01A2D8  000000A0  98 EF 9 0C8            LM    R14,R15,8(R9)
01A2DC  000000A4  58 79 C 098            L     R7,152(R9,R0)
01A2E0  000000A8  60 27 C 000            STD   F2,0(R7,R0)
01A2E4  000000AC  58 6B C 014            L     R6,20(R11,R0)
01A2E8  000000B0  10 76                  LPR   R7,R6
01A2EA  000000B2  54 6D 0 164            N     R6,356(R13,R0)
01A2EE  000000B6  90 67 D 118            STM   R6,R7,280(R13)
01A2F2  000000BA  96 4E D 118            OI    280(R13),X'4E'
01A2F6  000000BE  28 00                  SDR   F0,F0
01A2F8  000000C0  6A 0D C 118            AD    F0,280(R13,R0)
01A2FC  000000C4  2D 20                  DDR   F2,F0
01A2FE  000000C6  60 29 C 050            STD   F2,80(R9,R0)
01A302  000000CA  58 6B C 000            L     R6,0(R11,R0)
```

```
01A306  0000CE   50 69 C 098          ST    R6,152(R9,R0)
01A30A  0000D2   58 6B C 014          L     R6,20(R11,R0)
01A30E  0000D6   50 69 C 08C          ST    R6,128(R9,R0)
01A312  0000DA   68 C9 C 048          LD    F0,72(R9,R0)
01A316  0000DE   20 00                LPDR  F0,F0
01A318  0000E0   69 C9 C 028          CD    F0,40(R9,R0)
01A31C  0000E4   47 CC C 000          BC    0,0(R0,R0)
01A320  0000E8   47 BF C 0FE          BC    11,254(R15,R0)
01A324  0000EC   58 19 C 05C          L     R1,92(R9,R0)
01A328  0000F0   5A 1B C 00C          A     R1,0(R11,R0)
01A32C  0000F4   18 21                LR    R2,R1
01A32E  0000F6   47 CC C 000          BC    0,0(R0,R0)
01A332  0000FA   47 FF C 102          BC    15,258(R15,R0)
01A336  0000FE   58 2B C 000          L     R2,0(R11,R0)
01A33A  000102   50 29 C 05C          ST    R2,92(R9,R0)
01A33E  000106   59 29 C 038          C     R2,56(R9,R0)
01A342  00010A   47 C0 C 000          BC    0,0(R0,R0)
01A346  00010E   47 2F C 26C          BC    2,620(R15,R0)
01A34A  000112   58 19 C 018          L     R1,24(R9,R0)
01A34E  000116   58 69 C 080          L     R6,128(R9,R0)
01A352  00011A   50 61 C 028          ST    R6,40(R1,R0)
01A356  00011E   98 EF 9 030          LM    R14,R15,48(R9)
01A35A  000122   05 1F                BALR  R1,R15
01A35C  000124   98 EF 9 008          LM    R14,R15,8(R9)
01A360  000128   60 29 C 040          STD   F2,64(R9,R0)
01A364  00012C   58 6B C 000          L     R6,0(R11,R0)
01A368  000130   50 69 C 0A4          ST    R6,164(R9,R0)
01A36C  000134   58 69 C 058          L     R6,88(R9,R0)
01A370  000138   50 69 C 0A8          ST    R6,168(R9,R0)
01A374  00013C   58 6B C 000          L     R6,0(R11,R0)
01A378  000140   50 69 C 088          ST    R6,136(R9,R0)
01A37C  000144   58 29 C 088          L     R2,136(R9,R0)
01A380  000148   59 29 C 0A8          C     R2,168(R9,R0)
01A384  00014C   47 00 C 000          BC    0,0(R0,R0)
01A388  000150   47 2F C 1B2          BC    2,434(R15,R0)
01A38C  000154   50 29 C 088          ST    R2,136(R9,R0)
01A390  000158   D2 C7 9 048  9 040    MVC   72(8,R9),64(R9)
01A396  00015E   58 39 C 060          L     R3,96(R9,R0)
01A39A  000162   58 19 C 088          L     R1,136(R9,R0)
01A39E  000166   5C C9 C 07C          M     R0,124(R9,R0)
01A3A2  00016A   1A 31                AR    R3,R1
01A3A4  00016C   68 09 C 040          LD    F0,64(R9,R0)
01A3A8  000170   6A 03 C 000          AD    F0,0(R3,R0)
01A3AC  000174   58 6B C 014          L     R6,20(R11,R0)
01A3B0  000178   1C 76                LPR   R7,R6
01A3B2  00017A   54 6D C 164          N     R6,356(R13,R0)
01A3B6  00017E   90 67 D 118          STM   R6,R7,280(R13)
01A3BA  000182   96 4E D 118          OI    280(R13),X'4E'
01A3BE  000186   2B 22                SDR   F2,F2
01A3C0  000188   6A 2D C 118          AD    F2,280(R13,R0)
01A3C4  00018C   2D 02                CDR   F0,F2
01A3C6  00018E   60 C9 C 040          STD   F0,64(R9,R0)
01A3CA  000192   58 29 C 06C          L     R2,96(R9,R0)
01A3CE  000196   58 59 C 088          L     R5,136(R9,R0)
01A3D2  00019A   5C 49 C 07C          M     R4,124(R9,R0)
01A3D6  00019E   1A 25                AR    R2,R5
01A3D8  0001A0   D2 07 2 000  9 048    MVC   0(8,R2),72(R9)
01A3DE  0001A6   58 29 C 088          L     R2,136(R9,R0)
```

```
01A3E2  0001AA   98 01 5 0A4          LM    R0,R1,164(R9)
01A3E6  0001AE   87 20 F 154          BXLE  R2,R0,340(R15)
01A3EA  0001B2   68 C9 0 040          LD    F0,64(R9,R0)
01A3EE  0001B6   20 C0                LPDR  F0,F0
01A3F0  0001B8   58 39 C 060          L     R3,96(R9,R0)
01A3F4  0001BC   58 19 0 058          L     R1,88(R9,R0)
01A3F8  0001C0   5C C9 C 07C          M     R0,124(R9,R0)
01A3FC  0001C4   1A 31                AR    R3,R1
01A3FE  0001C6   68 23 C 000          LD    F2,0(R3,R0)
01A402  0001CA   20 22                LPDR  F2,F2
01A404  0001CC   29 02                CDR   F0,F2
01A406  0001CE   41 50 C 000          LA    R5,0(R0,R0)
01A40A  0001D2   47 C0 C 000          BC    0,0(R0,R0)
01A40E  0001D6   47 BF C 1DE          BC    11,478(R15,R0)
01A412  0001DA   41 50 C 001          LA    R5,1(R0,R0)
01A416  0001DE   58 29 C 058          L     R2,88(R9,R0)
01A41A  0001E2   59 2B C 010          C     R2,16(R11,R0)
01A41E  0001E6   41 40 C 000          LA    R4,0(R0,R0)
01A422  0001EA   47 C0 C 000          BC    0,0(R0,R0)
01A426  0001EE   47 BF C 1F6          BC    11,502(R15,R0)
01A42A  0001F2   41 40 C 001          LA    R4,1(R0,R0)
01A42E  0001F6   14 54                NR    R5,R4
01A430  0001F8   12 55                LTR   R5,R5
01A432  0001FA   47 00 C 000          BC    0,0(R0,R0)
01A436  0001FE   47 8F C 24A          BC    8,586(R15,R0)
01A43A  000202   58 19 C 058          L     R1,88(R9,R0)
01A43E  000206   5A 1B C 000          A     R1,0(R11,R0)
01A442  00020A   50 19 C 058          ST    R1,88(R9,R0)
01A446  00020E   58 39 C 060          L     R3,96(R9,R0)
01A44A  000212   58 19 C 058          L     R1,88(R9,R0)
01A44E  000216   5C C9 C 07C          M     R0,124(R9,R0)
01A452  00021A   1A 31                AR    R3,R1
01A454  00021C   D2 07 3 0CC  9 040   MVC   0(8,R3),64(R9)
01A45A  000222   58 6B 0 014          L     R6,20(R11,R0)
01A45E  000226   10 76                LPR   R7,R6
01A460  000228   54 6D C 164          N     R6,356(R13,R0)
01A464  00022C   90 67 D 118          STM   R6,R7,280(R13)
01A468  000230   96 4E D 118          OI    280(R13),X'4E'
01A46C  000234   2B 44                SDR   F4,F4
01A46E  000236   6A 4D C 118          AD    F4,280(R13,R0)
01A472  00023A   68 69 C 040          LD    F6,64(R9,R0)
01A476  00023E   2D 64                CDR   F6,F4
01A478  000240   28 26                LDR   F2,F6
01A47A  000242   47 00 C 000          BC    0,0(R0,R0)
01A47E  000246   47 FF C 24E          BC    15,590(R15,R0)
01A482  00024A   68 29 C 040          LD    F2,64(R9,R0)
01A486  00024E   60 29 C 048          STD   F2,72(R9,R0)
01A48A  000252   68 09 C 050          LD    F0,80(R9,R0)
01A48E  000256   2A 02                ADR   F0,F2
01A490  000258   60 09 C 050          STD   F0,80(R9,R0)
01A494  00025C   58 29 C 080          L     R2,128(R9,R0)
01A498  000260   5A 29 C 098          A     R2,152(R9,R0)
01A49C  000264   50 29 C 080          ST    R2,128(R9,R0)
01A4A0  000268   47 F0 F 0DA          BC    15,218(R0,R15)
01A4A4  00026C   68 29 C 050          LD    F2,80(R9,R0)
01A4A8  000270   58 90 5 000          L     R9,0(R0,R9)
01A4AC  000274   58 10 5 004          L     R1,4(R0,R9)
01A4B0  000278   07 F1                BCR   R15,R1
```

```
01A4B8  00000000  47 F0 F 00E              BC    15,14(R0,R15)
01A4BC  00000004  0CCCCCCC                 DC    X'00000000'
01A4C0  00000008  0C0CCC4C                 DC    X'00000040'
01A4C4  0000000C  000D                     DC    X'000D'
01A4C6  0000000E  45 80 D 3C0              BAL   R8,768(R0,R13)
01A4CA  00000012  41 79 C 028              LA    R7,40(R9,R0)
01A4CE  00000016  50 79 C 014              ST    R7,20(R9,R0)
01A4D2  0000001A  D2 07 7 00C  E 000       MVC   0(8,R7),0(R14)
01A4D8  00000020  50 97 C 0C8              ST    R9,8(R7,R0)
01A4DC  00000024  58 E7 C 004              L     R14,4(R7,R0)
01A4E0  00000028  90 EF 9 0C8              STM   R14,R15,8(R9)
01A4E4  0000002C  50 29 C 038              ST    R2,56(R9,R0)
01A4E8  00000030  1B CC                    SR    R0,R0
01A4EA  00000032  50 C9 C 03C              ST    R0,60(R9,R0)
01A4EE  00000036  58 CB C 004              L     R0,4(R11,R0)
01A4F2  0000003A  50 C2 C 000              ST    R0,0(R2,R0)
01A4F6  0000003E  50 52 C 004              ST    R5,4(R2,R0)
01A4FA  00000042  58 CB C 0C8              L     R0,8(R11,R0)
01A4FE  00000046  50 C2 C 014              ST    R0,20(R2,R0)
01A502  0000004A  58 0B C 00C              L     R0,12(R11,R0)
01A506  0000004E  50 02 C 018              ST    R0,24(R2,R0)
01A50A  00000052  58 CB C 000              L     R0,0(R11,R0)
01A50E  00000056  58 29 C 038              L     R2,56(R9,R0)
01A512  0000005A  50 02 0 01C              ST    R0,28(R2,R0)
01A516  0000005E  58 0B C 010              L     R0,16(R11,R0)
01A51A  00000062  58 29 C 038              L     R2,56(R9,R0)
01A51E  00000066  50 C2 C 02C              ST    R0,32(R2,R0)
01A522  0000006A  41 22 C 0C8              LA    R2,8(R2,R0)
01A526  0000006E  58 F0 D 618              L     R15,1560(R0,R13)
01A52A  00000072  05 1F                    BALR  R1,R15
01A52C  00000074  0003                     DC    X'0003'
01A52E  00000076  98 EF 9 008              LM    R14,R15,8(R9)
01A532  0000007A  58 29 C 038              L     R2,56(R9,R0)
01A536  0000007E  58 52 C 004              L     R5,4(R2,R0)
01A53A  00000082  18 75                    LR    R7,R5
01A53C  00000084  5C 62 C 0C0              M     R6,12(R2,R0)
01A540  00000088  5A 79 C 03C              A     R7,60(R9,R0)
01A544  0000008C  41 70 7 0C7              LA    R7,7(R0,R7)
01A548  00000090  88 70 0 003              SRL   R7,3(R0)
01A54C  00000094  89 70 0 003              SLL   R7,3(R0)
01A550  00000098  50 79 C 03C              ST    R7,60(R9,R0)
01A554  0000009C  58 29 C 03C              L     R2,60(R9,R0)
01A558  000000A0  58 90 9 000              L     R9,0(R0,R9)
01A55C  000000A4  58 1C 9 0C4              L     R1,4(R0,R9)
01A560  000000A8  07 F1                    BCR   R15,R1
```

```
FBCTE  AT 01AJA8
RTCC00 AT 01A110
RTCC01 AT 01A238
RTCC02 AT 01A4B8
RTCC03 AT 01A178
     TEMPS DE CHARGEMENT      536,76MILLISECONDES
```

EXECUTION ALGOL68 EULER2

TEMPS D'EXECUTION 0,43MILLISECONDES

DUMP ALGOL68 EULER2

CONTENU DE LA PILE

NUMERO D'UNITE DE LA ROUTINE= 0 ADRESSE D'APPEL 018984

```
BASEPILE  0001A0C8  ENVPRECE  A0C18980  BASEPROG  0001A110  TETETAS  00018170
BASESYST  00017ZA0  BASEGLOB  0001A0C8  BASELTES  0001A0A8  BASEM4K  0001A564
```

REGISTRES

```
GR 0-7  00017600  4001A2AE  5001898E  5J018998  00000001  00000001  000180C0  00000002
GR 8-F  00018070  0001A0C8  0001A564  0001A0A8  0001A0C8  00017ZA0  A0018980  0001A110
FPREGS  4E18A026  95D8A572  4E18A026  95C8A572  41200000  C0CC0000  3C31A6C3  1A603F00
```

DONNEES DE LIAISON

```
BACKLINK  FBFBFBFB  RTURNADR  60C1A16C  DISPLPTR  0001A568  STACKPTR  0001A570
REFMAPC   00000000  BASERCUT  00C1A110
```

PARTIE STATIQUE

```
01A0C8  FBFBFBFB  0001A16C  90018966  0001A110  C5E4D940  0001A568  0001A570  00000000  *...............EUR ...........*
01A1E8  D7406709  0000C4C2  0001A0C8  0001A238  C0B17231  802F48C3  0001A238  0001A488  *P GR..DB...H..........C........*
01A1C8  0001A178  09C540E3                                                               *....KE T.00............N.......*
```