Etude de la compilation de BASEL, langage de la famille d'Algol 68

Victoria Bajar

N° d'ordre

THESE

présentée à

L'Université Scientifique et Médicale de Grenoble

pour obtenir

Le grade de Docteur de troisième cycle

"Informatique"
par

VICTORIA R. BAJAR

ETUDE DE LA COMPILATION DE BASEL, LANGAGE DE LA FAMILLE D'ALGOL 68

Soutenue le 12 FEVRIER 1973 devant la commission d'examen

MM. Bernard VAUQUOIS Président

Jean-Claude BOUSSARD Rapporteur

Michaël GRIFFITHS Examinateur

Philippe JORRAND Invité

UNIVERSITE SCIENTIFIQUE

ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

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

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BEZES Henri	Chirurgie générale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BONNIER Etienne	Electrochimie Electrométallurgie
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BRAVARD Yves	Géographie
	BRISSONNEAU Pierre	Physique du Solide
	BUYLE-BODIN Maurice	Electronique
	CABANAC Jean	Pathologie chirurgicale
	CABANEL Guy	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Oto-Rhino-Laryngologie
	CHATEAU Robert	Thérapeutique
	CHENE Marcel	Chimie papetière
	COEUR André	Pharmacie chimique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	FELICI Noël	Electrostatique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures

MM. GASTINEL Noël	Analyse numérique
GERBER Robert	Mathématiques Pures
GIRAUD Pierre	Géologie
KLEIN Joseph	Mathématiques Pures
Mme KOFLER Lucie	Botanique et Physiologie végétale
MM. KOSZUL Jean-Louis	Mathématiques Pures
KRAVTCHENKO Julien	Mécanique
KUNTZMANN Jean	Mathématiques Appliquées
LACAZE Albert	Thermodynamique
LACHARME Jean	Biologie végétale
LATREILLE René	Chirurgie générale
LATURAZE Jean	Biochimie pharmaceutique
LAURENT Pierre	Mathématiques Appliquées
LEDRU Jean	Clinique médicale B
LLIBOUTRY Louis	Géophysique
LOUP Jean	Géographie
Mle LUTZ Elisabeth	Mathématiques Pures
MALGRANGE Bernard	Mathématiques Pures
MALINAS Yves	Clinique obstétricale
MARTIN-NOEL Pierre	Seméiologie médicale
MASSEPORT Jean	Géographie
MAZARE Yves	Clinique médicale A
MICHEL Robert	Minéralogie et Pétrographie
MOURIQUAND Claude	Histologie
MOUSSA André	Chimie nucléaire
NEEL Louis	Physique du Solide
OZENDA Paul	Botanique
PAUTHENET René	Electrotechnique
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRET René	Servomécanismes
PILLET Emile	Physique industrielle
RASSAT André	Chimie systématique
RENARD Michel	Thermodynamique
REULOS René	Physique industrielle
RINALDI Renaud	Physique
ROGET Jean	Clinique de pédiatrie et de puériculture
SANTON Lucien	Mécanique
SEIGNEURIN Raymond	Microbiologie et Hygiène
SENGEL Philippe	Zoologie
SILBERT Robert	Mécanique des fluides
SOUTIF Michel	Physique générale
TANCHE Maurice	Physiologie
TRAYNARD Philippe	Chimie générale
VAILLAND François	Zoologie
VAUQUOIS Bernard	Calcul électronique
Mme VERAIN Alice	Pharmacie galénique
M. VERAIN André	Physique
Mme VEYRET Germaine	Géographie
MM. VEYRET Paul	Géographie
VIGNAIS Pierre	Biochimie médicale
YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM. BULLEMER Bernhard	Physique
RADHAKRISHNA Pidatala	Thermodynamique

PROFESSEURS SANS CHAIRE

MM. AUBERT Guy	Physique
BEAUDOING André	Pédiatrie
BERTRANDIAS Jean-Paul	Mathématiques Appliquées
BIARES Jean-Pierre	Mécanique
BONNETAIN Lucien	Chimie minérale
Mme BONNIER Jane	Chimie générale
MM. CARLIER Georges	Biologie végétale
COHEN Joseph	Electrotechnique
COUMES André	Radioélectricité
DEPASSEL Roger	Mécanique des Fluides
DEPORTES Charles	Chimie minérale
DESRE Pierre	Métallurgie
DOLIQUE Jean-Michel	Physique des Plasmas
GAUTHIER Yves	Sciences biologiques
GEINDRE Michel	Electroradiologie
GIDON Paul	Géologie et Minéralogie
GLENAT René	Chimie organique
HACQUES Gérard	Calcul numérique
JANIN Bernard	Géographie
Mme KAHANE Josette	Physique
MM. MULLER Jean-Michel	Thérapeutique
PERRIAUX Jean-Jacques	Géologie et minéralogie
POULOUJADOFF Michel	Electrotechnique
REBECQ Jacques	Biologie (CUS)
REVOL Michel	Urologie
REYMOND Jean-Charles	Chirurgie générale
ROBERT André	Chimie papetière
SARRAZIN Roger	Anatomie et chirurgie
SARROT-REYNAULD Jean	Géologie
SIBILLE Robert	Construction Mécanique
SIROT Louis	Chirurgie générale
Mme SOUTIF Jeanne	Physique générale
M. VALENTIN Jacques	Physique nucléaire

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

Mle AGNIUS-DELORD Claudine	Physique pharmaceutique
ALARY Josette	Chimie analytique
MM. AMBLARD Pierre	Dermatologie
AMBROISE-THOMAS Pierre	Parasitologie
ARMAND Yves	Chimie
BEGUIN Claude	Chimie organique
BELORIZKY Elie	Physique
BENZAKEN Claude	Mathématiques Appliquées
Mme BERTRANDIAS Françoise	Mathématiques Pures
MM. BLIMAN Samuel	Electronique (EIE)
BLOCH Daniel	Electrotechnique
Mme BOUCHE Liane	Mathématiques (CUS)
MM. BOUCHET Yves	Anatomie
BOUSSARD Jean-Claude	Mathématiques Appliquées
BOUVARD Maurice	Mécanique des Fluides
BRIERE Georges	Physique expérimentale
BRODEAU François	Mathématiques (IUT B)
BRUGEL Lucien	Energétique
BUISSON Roger	Physique
BUTEL Jean	Orthopédie
CHAMBAZ Edmond	Biochimie médicale
CHAMPETIER Jean	Anatomie et organogénèse

MM.	CHIAVERINA Jean	Biologie appliquée (EFP)
	CHIBON Pierre	Biologie animale
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique
	CROUZET Guy	Radiologie
	DURAND Francis	Métallurgie
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GAVEND Michel	Pharmacologie
	GENSAC Pierre	Botanique
	GERMAIN Jean-Pierre	Mécanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques Appliquées
	GROULADE Joseph	Biochimie médicale
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Médecine préventive
	IDELMAN Simon	Physiologie animale
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	JOLY Jean-René	Mathématiques Pures
	JOUBERT Jean-Claude	Physique du Solide
	JULLIEN Pierre	Mathématiques Pures
	KAHANE André	Physique générale
	KUHN Gérard	Physique
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LANCIA Roland	Physique atomique
	LE JUNTER Noël	Electronique
	LEROY Philippe	Mathématiques
	LOISEAUX Jean-Marie	Physique Nucléaire
	LONGEQUEUE Jean-Pierre	Physique Nucléaire
	LUU DUC Cuong	Chimie Organique
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MAYNARD Roger	Physique du Solide
	MICOUD Max	Maladies infectieuses
	MOREAU René	Hydraulique (INP)
	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PERRIN Louis	Pathologie expérimentale
	PFISTER Jean-Claude	Physique du Solide
	PHELIP Xavier	Rhumatologie
Mle	PIERY Yvette	Biologie animale
MM.	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RICHARD Lucien	Botanique
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	ROUGEMONT (DE) Jacques	Neuro-Chirurgie
	STIEGLITZ Paul	Anesthésiologie

MM. STOEBNER Pierre	Anatomie pathologique
VAN CUTSEM Bernard	Mathématiques Appliquées
VEILLON Gérard	Mathématiques Appliquées (INP)
VIALON Pierre	Géologie
VOOG Robert	Médecine interne
VROUSSOS Constantin	Radiologie
ZADWORNY François	Electronique

MAITRES DE CONFERENCES ASSOCIES

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

CHARGES DE FONCTIONS DE MATIRES DE CONFERENCES

Mme BERIEL Hélène	Physiologie
Mme RENAUDET Jacqueline	Microbiologie

Fait le 8 MARS 1972.

Mes rapports avec l'Université de Grenoble ont commencé lors du voyage en Argentine de Monsieur le Professeur VAUQUOIS dont l'intérêt et l'appui ont permis que mon voyage à Grenoble fût possible. Je le remercie très sincèrement de m'avoir fait l'honneur de présider le jury de ma thèse.

Je dois à Monsieur le Professeur BOUSSARD mon séjour à Grenoble. Cette étude a été réalisée dans son équipe. Je suis reconnaissante de l'intérêt, la critique précise et le jugement sagace qu'il a porté à tout moment à mon travail. Sa direction a été très fructueuse pour moi. Je le remercie en outre d'avoir résolu quelques situations difficiles pendant mon séjour.

Je remercie Monsieur GRIFFITHS, Maître de Conférences à l'Université Scientifique et Médicale de Grenoble, d'avoir bien voulu faire partie du jury de ma thèse.

Monsieur Ph. JORRAND, du Centre Scientifique I. B. M. de Grenoble, est l'auteur de Basel, le langage qui fait l'objet de cette étude. Je le remercie vivement de ses conseils et de son aide qui m'ont été précieux tout au long de celle-ci, ainsi que de m'avoir facilité au maximum les conditions matérielles de travail.

La réalisation d'un travail étant d'une certaine manière la synthèse des efforts de nombreuses personnes, je remercie tous ceux qui m'ont apporté leur aide. En particulier Bernard MAILLOT et Guy TASSART qui ont essayé de chasser tout hispanisme du texte.
Je remercie également l'équipe des opérateurs du Laboratoire d'Informatique pour leur collaboration lors des travaux nocturnes et en fin tous ceux qui ont contribué à la réalisation matérielle de cette thèse.

Victoria BAJAR

A mes parents.

TABLE DETAILLEE DES CHAPITRES

Chapitre 1 : LE LANGAGE BASEL -

Chapitre 2 : LE TRAITEMENT DES MODES -

Chapitre 3 : LES MODIFICATIONS -

Chapitre 4 : ARCHITECTURE GENERALE DU COMPILATEUR -

INTRODUCTION

Le travail présenté ici est une étude des problèmes fondamentaux posés par la compilation de BASEL [JO] [JH], langage de programmation dans un style analogue à celui d'Algol 68. Comme Algol 68, BASEL est un langage construit autour de la notion de mode. Chaque valeur qui peut faire l'objet d'un traitement dans un programme BASEL a un mode, et c'est ce mode qui détermine les usages que l'on peut faire de la valeur considérée. C'est pourquoi, au cours de la compilation d'un programme BASEL, il est nécessaire de prévoir le mode des valeurs qui seront produites par chaque expression au moment de l'exécution, afin de contrôler si ces valeurs sont à leur tour utilisées correctement à l'intérieur d'expressions plus larges.

Au chapitre 1, on présentera le langage BASEL, puis, dans le chapitre 2 on décrira les divers mécanismes qui ont été étudiés pour traiter les modes pendant la compilation d'un programme BASEL. C'est ainsi que l'on choisira une représentation interne pour les modes et qu'on définira un algorithme pour décider si deux représentations sont une image du même mode. On cherchera également à obtenir, pour un mode donné, une représentation minimale. L'équivalence de deux représentations et l'obtention d'une représentation minimale seront abordées de façon formelle avant de proposer des algorithmes. Les techniques de traitement des modes présentées dans ce chapitre sont à rapprocher de travaux effectués sur Algol 68. En effet, Koster [KO] a proposé une méthode pour comparer deux modes d'Algol 68, mais il n'a pas couvert l'ensemble des problèmes liés au traitement des modes, particulièrement en ce qui concerne la réduction et l'unicité des représentations des modes. D'autre part, le travail de Marché [MA], bien qu'étant orienté, comme celui qui est présenté ici, vers la recherche de techniques d'implémentation, ne couvre pas non plus l'ensemble des problèmes. De plus, le traitement des modes étant un processus complexe, il est nécessaire de commencer par l'aborder d'une façon formelle, ce qui ne semble pas avoir été fait dans [MA].

Ensuite, dans le chapitre 3, on étudiera les divers cas où BASEL permet de calculer une valeur d'un certain mode dans un contexte qui demande en fait, une valeur d'un autre mode : la première valeur devra être transformée en une valeur du second mode. Ce mécanisme, appelé modification, généralise celui des conversions, de la même façon que la notion de mode généralise la notion de type. L'étude formelle et algorithmique qui en sera faite permettra d'isoler les principes généraux à partir desquels le compilateur BASEL sera capable de choisir dans tous les cas les modifications auxquelles il devra faire appel.

Enfin dans le chapitre 4, on décrira la structure générale du compilateur BASEL et la façon dont il fait appel aux divers algorithmes qui ont été developpés dans les chapitres précédents.

PLAN GENERAL

CHAPITRE I

L E L A N G A G E B A S E L

I - INTRODUCTION -

BASEL [JO] [JH] est un langage de programmation d'un style analogue à celui d'Algol 68 [WI] ; il a à peu près les mêmes composants fondamentaux [WI, 0 à 8] mais une structure plus simple. Tout au long de cet exposé, on notera les points de rapprochement ou éventuellement les divergences entre les deux langages au moyen de références.

BASEL permet d'écrire des programmes exécutables sur les ordinateurs actuels, au moyen d'un compilateur qui peut se concevoir de façon très systématique, fait qui le rend relativement simple et clair (voir Ch.4). Toutes les erreurs syntaxiques et beaucoup d'erreurs de nature sémantique peuvent être signalées à la compilation. Enfin, comme on le verra plus loin (voir Ch.4), un programme BASEL peut toujours être compilé en une seul passage.

I.1 - Quelques caractéristiques de BASEL -

I.1.1 - Les modes -

Comme Algol 68, BASEL fait usage de la notion de mode qui est une généralisation de la notion de type existant en Algol 60. En BASEL chaque objet qui représente une valeur a toujours un mode, et un seul. Il existe un ensemble de modes qui font partie du langage et en plus, un mécanisme de définition des modes qui permet de construire autant de modes différents que l'on veut ; ainsi, l'ensemble des modes que l'on peut utiliser est infini, bien qu'il reste fini dans un programme donné. La vérification des modes des objets d'un programme BASEL se fait au moment de la compilation et il n'existe pas de traitement des modes au moment de l'exécution, sauf dans le cas de vérification du mode courant de la valeur d'un objet de mode union (voir Ch.1, II.2) [WI, 0.1.4.1].

D'après la syntaxe de BASEL, on peut faire l'analyse d'un programme d'une façon indépendante des modes des objets, mais il existe un ensemble de règles sémantiques qui permet de décider si un programme est correct ou non en ce qui concerne l'utilisation des modes [WI, O. 1. 4. 3].

I.1.2 - La portée -

La vérification de portée des objets est statique, elle se fait entièrement au moment de la compilation [WI, O. 1. 4. 2].

I.1.3 - Les valeurs -

Il en existe quatre catégories :

1 - VALEUR SIMPLE : une valeur simple est une valeur entière, ou une valeur réelle, ou une valeur booléenne ou une valeur caractère, de mode int, real, bool ou char respectivement.

2 - ADRESSE : une valeur qui identifie une certaine adresse ou emplacement dans la mémoire libre (Voir Ch. 1, I.1.4).

3 - ENSEMBLE : il existe trois types d'ensemble ,

3a - TUPLE : ensemble ordonné de n valeurs qui se manipule comme un tout. Les modes des n objets composants peuvent être identiques ou différents.

3b - STRUCTURE ou VALEUR STRUCTUREE : ensemble ordonné de n valeurs appelées champs. Chaque champ est identifié par un sélecteur de champ, et il peut être manipulé séparement des autres. Les modes des champs peuvent être identiques ou différents [WI, 2. 2. 3. 2].

3c - SEQUENCE : ensemble ordonné de n valeurs, toutes de même mode. Il est possible de manipuler le tout et de manipuler une valeur ou un sous-ensemble de valeurs d'une séquence. Le choix d'une valeur ou d'un sous-ensemble se fait au moyen d'indices [WI, 2. 2. 3. 3].

4 - PROCEDURES : peuvent être sans ou avec paramètres, elles peuvent donner ou non un résultat. En plus d'être définies et appelées, les procédures sont elles-mêmes des valeurs, et peuvent être manipulées comme telles.

I.1.4 - L'Allocation de Mémoire -

BASEL suppose une mémoire constituée d'une pile et d'une mémoire libre. Dans la pile sont rangées les valeurs des variables utilisées et les résultats des expressions calculées. Dans la mémoire libre se trouvent les objets repérés par les valeurs adresses et les valeurs composants des séquences.

L'allocation de la mémoire libre est entièrement dynamique. La gestion de la mémoire libre implique éventuellement l'exécution des algorithmes de "ramasse-miettes" pendant l'exécution d'un programme.

I.1.5 - Les déclarations -

Il existe des déclarations de variables et de modes. Les premières servent à définir le nom et le mode des variables à utiliser. Les autres servent à définir des symboles et des modes représentés par ces symboles.

I.1.6 - Les instructions -

Il existe deux types d'instructions : les expressions, qui donnent en général une valeur, parmi lesquelles se trouve l'instruction d'affectation ; et les instructions de transfert et d'opérations sur fichiers qui ne donnent pas de **valeur.**

I.1.7 - L'élaboration d'un programme BASEL -

L'élaboration s'effectue "en série" ou "dans un ordre indéfini". On retrouve ici, dans une certaine mesure les notions "serial" et "collateral" [WI, O. 2. 4] d'Algol 68, mais avec des différences, comme on le verra par la suite.

I.2 - Description du langage -

Les différentes notions de BASEL seront introduites au moyen de descriptions en français accompagnées d'exemples. La syntaxe figure dans l'Appendice I.

I.2.1 - Alphabet des terminaux -

C'est l'ensemble constitué par l'union des ensembles :

C = {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T,
U, V, W, X, Y, Z,
a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t,
u, v, w, x, y, z,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
+, -, /, *, (,), [,], |, :, <, =, >, ≠, ,, ., ;, _, ',
"", ␣ }

et

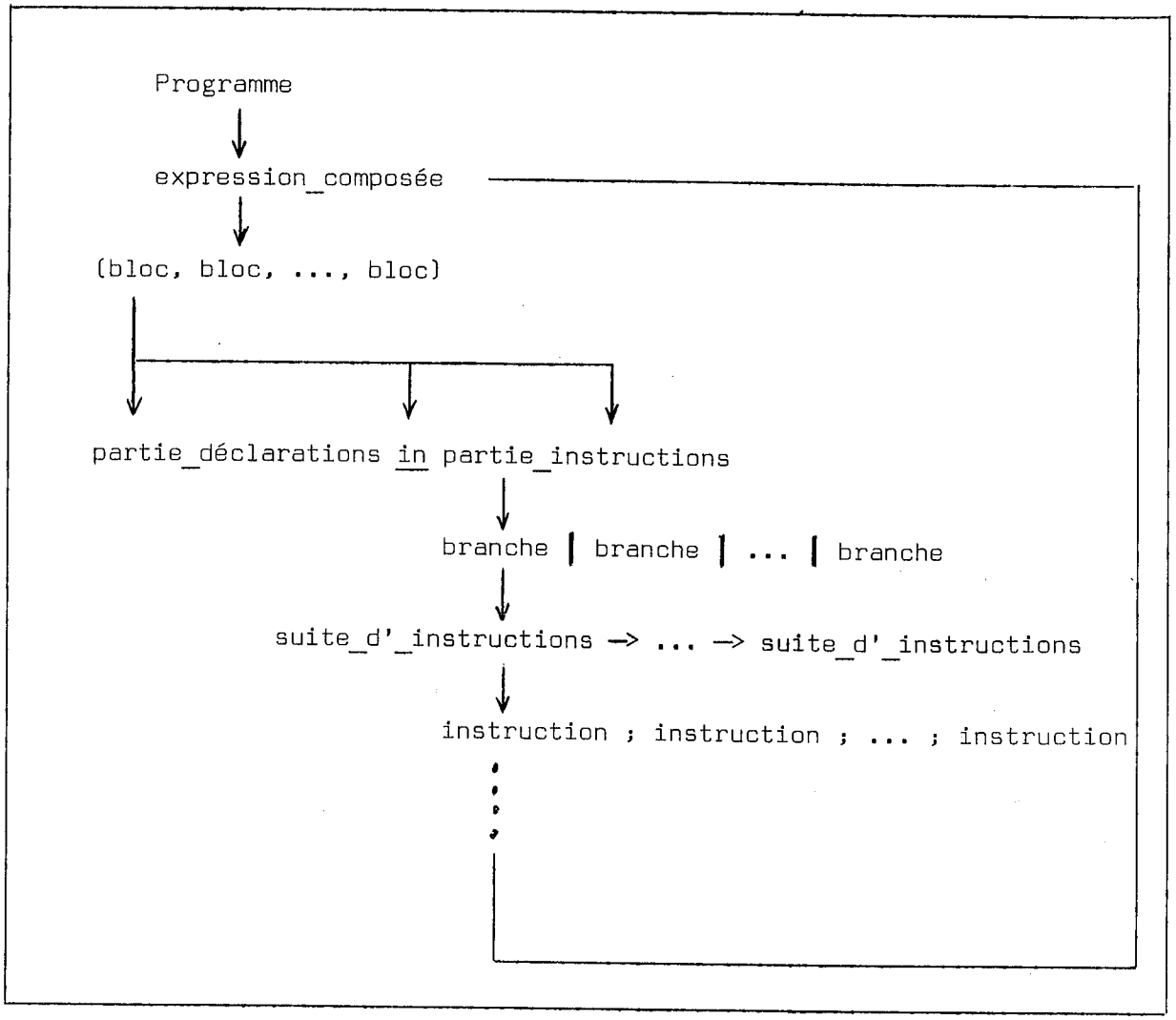
S = {int, real, bool, char, ptr, seq, tuple, struct, union, proc,
none, is, rep, means, generic, mustbe, in, of, times, alloc,
set, same, change, length, or, and, not, eq, neq, less, leq,
grt, geq, plus, minus, mul, div, idiv, rem, exp, odd,
abs, sign, round, ent, float, iread, rread, bread, cread,
write, input, output, close, goto, true, false, →, |,
,, ;, (,), [,]}.

Les éléments de S s'appellent symboles du langage.

Certains caractères appartenant à C apparaissent aussi dans S car ils ont une signification spéciale. Le blanc est significatif dans le cas où il n'y a pas d'autre symbole séparateur.

1.2.2 - Structure syntaxique d'un programme BASEL -

Un programme BASEL est constitué de blocs qui peuvent être imbriqués ou adjacents, d'après une structure que l'on peut schématiser de la façon suivante :



Il est intéressant de noter qu'une instruction peut elle-même être une expression_composée, ce qui donne une idée de la récursivité de la syntaxe de BASEL (App. I).

Une expression_composée peut n'avoir qu'un seul bloc, ou même être vide.

Chaque bloc contient une partie_instructions qui peut être éventuellement précédée d'une partie_déclarations. Les objets déclarés dans la partie_déclarations sont utilisables dans la partie_instructions, et leur portée est définie de façon analogue à celle qui existe en Algol 60 (sauf pour les variables globales utilisées dans les procédures).

I.2.3 - Elaboration d'un programme BASEL -

Un programme BASEL, écrit d'après les règles de la syntaxe, définit une suite d'actions qu'un ordinateur est capable d'accomplir. Comme en Algol 68, on appelle élaboration du programme le fait d'accomplir cette suite d'actions [LY, 1. 1. 1]. L'élaboration d'un programme BASEL est l'élaboration d'une expression composée.

L'élaboration d'une expression composée consiste en l'élaboration de chacun des blocs qui la composent, l'un après l'autre, en série, suivant l'ordre dans lequel ils se trouvent (cela ne correspond ni à la notion "collateral" [WI, 2. 2. 5. b] d'Algol 68, ni à la notion "serial" [WI, 2. 2. 5. a] comme on verra plus loin).

L'élaboration d'un bloc consiste d'abord en l'élaboration de sa partie_déclarations, si elle existe et ensuite en l'élaboration de sa partie_instructions.

L'élaboration de la partie déclarations est l'élaboration de chacune des déclarations qui la composent, sans aucune spécification d'ordre.

L'élaboration de la partie instructions consiste à élaborer ses instructions séparées par le symbole ;, l'une_après_l'autre, c'est-à-dire : en série [WI, 2. 2. 5. a], et jusqu'à la rencontre d'un des symboles qui agissent sur la terminaison de l'élaboration d'un bloc. Ces symboles sont les symboles →, | , ,,) et goto, chacun possédant une signification que l'on précisera par la suite. Comme résultat de l'élaboration d'un bloc on a en général une valeur : la valeur de la dernière instruction élaborée ; parfois, par contre, le bloc n'a pas de valeur. Comme résultat de l'élaboration d'une expression_composée on a un ensemble ordonné (tuple, plus précisement) de valeurs : les valeurs de chacun de ses blocs composants. Cet ensemble est la valeur de l'expression composée. (La détermination d'une valeur pour l'expression_composée précise la différence avec la notion "serial" d'Algol 68 [WI, 2.2.5.a]).

Les symboles —>, |, ,,) et goto.

1 - SYMBOLE —> : il implique un test sur la dernière valeur élaborée, qui doit être booléenne. Si cette valeur est "vrai", l'élaboration continue à partir des instructions qui suivent le symbole —>. Si elle est "fausse", l'élaboration continue à partir des instructions se trouvant après le prochain symbole | dans le bloc. Il se peut d'après la syntaxe, qu'il n'existe pas de symbole |, cas où l'élaboration du bloc est terminée et le bloc n'a pas de valeur.

2 - SYMBOLE | : la rencontre du symbole | pendant l'élaboration du bloc fait terminer son élaboration, en gardant la dernière valeur élaborée comme celle du bloc. La valeur d'un bloc dans lequel il y a plusieurs branches est une valeur de mode union (voir Ch. 1, II.2) car il est impossible de connaître "a priori" le mode de la valeur du bloc, tant qu'il n'est pas élaboré.

3 - SYMBOLES , et) : indiquent la fin du bloc ; le symbole) indique de plus la fin de l'expression_composée ; l'élaboration du bloc étant terminée, la valeur du bloc est la dernière valeur élaborée.

4 - SYMBOLE goto : la rencontre d'un symbole goto suivi d'une étiquette qui appartient au bloc où se trouve le goto, est élaborée comme une instruction de transfert classique ; c'est-à-dire que l'élaboration du bloc continue à partir du point identifié par l'étiquette. Mais si l'étiquette n'appartient pas au bloc où se trouve le symbole goto, l'élaboration du bloc est terminée ; alors le bloc n'a pas de valeur du fait que l'instruction de transfert n'en calcule pas.

REMARQUE : à la compilation un bloc est considéré sans valeur si on s'aperçoit qu'à l'élaboration il existe une possibilité, sinon une certitude qu'il se termine sans valeur.

Ainsi, le bloc qui constitue l'expression_composée suivante est considéré sans valeur, car une des possibilités de terminaison ne donne pas de valeur :

(; ( ) | ; valeur | ; valeur)

↓

pas de valeur : expression_composée vide

Dans le cas particulier de la structure suivante :

(; goto L | ; valeur | ; valeur)

on considère (à la compilation) que le bloc a une valeur de mode union et n'est pas sans valeur. En effet, si l'étiquette L est dans le bloc, on ne sortira du bloc qu'en ayant calculé la valeur de l'une des deux autres branches. Si l'étiquette est en dehors du bloc, la syntaxe de BASEL est telle que toutes les valeurs déjà calculées pour les expressions qui pourraient utiliser une valeur de ce bloc sont supprimées au moment où le goto est exécuté : l'élaboration de ces expressions qui avait été commencée, est en quelques sorte "oubliée".

L'expression conditionnelle :

L'expression conditionnelle

if ... then ... else ... fi

d'Algol 68 [WI, 6. 4] s'écrit en BASEL au moyen de l'expression_composée

(→ |)

qui n'est qu'un cas particulier de la structure générale des expressions_composées.

Exemple :

(grt (i, j) → minus (i, j) | minus (j, i)) (Voir Ch. 1, V. 2 : signification des opérateus grt et minus).

II - LES MODES -

II.1 - Modes de base - Mode none -

Il y a quatre modes de base représentés par les symboles int, real, bool et char. Ce sont respectivement les modes des valeurs entières, des valeurs réelles, des valeurs booléennes et des caractères. Le mode none

est le mode auquel aucune valeur ne correspond ; il sert essentiellement à indiquer l'absence de valeur retournée, dans le mode d'une procédure sans résultat.

II.2 - Modes construits -

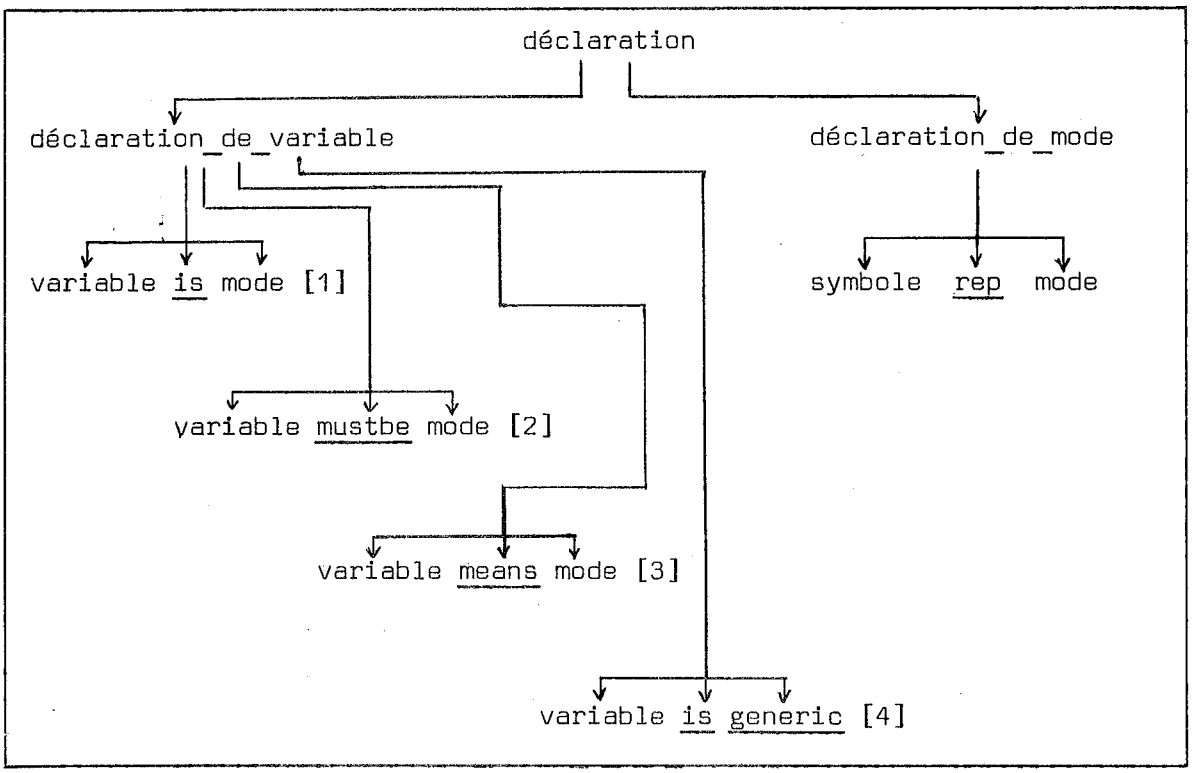
Six constructeurs de modes permettent de combiner des modes entre eux de façon à en construire de nouveaux. Ils sont représentés par les symboles ptr, seq, tuple, struct, union et proc. Le tableau ci-dessous définit les règles d'utilisation de chacun de ces constructeurs, et décrit brièvement la nature des valeurs qui ont les modes correspondants. Dans ce tableau : m, m1, m2, ... mn sont des modes, et s1, s2, ..., sn des identificateurs.

CONSTRUCTEUR	UTILISATION	NATURE DES VALEURS	EXEMPLES
ptr	ptr m	adresse ou emplacement d'un objet de mode m	ptr int
seq	seq m	séquence d'objets, tous de mode m, identifiés par indices	seq char
tuple	tuple [m1, m2, ..., mn].	ensemble ordonné de n objets, dont le i-ème est de mode mi.	tuple [int, bool]
struct	struct [m1 s1, m2 s2, ..., mn sn]	ensemble ordonné de n objets, dont le i-ème est de mode mi et est identifié par si.	struct [real r, real i]
union	union [m1, ..., mn]	objet dont la valeur peut être soit de mode m1, soit de mode m2, etc....	union [int, real, ptr int]

proc	proc [m1, m2, ... mn] m	procédure dont les n paramètres sont respectivement des modes m1, m2, ..., mn et dont le résultat est de mode m. La liste des modes mi peut être vide (procédure sans paramètres). m peut être none (procédure sans résultat).	proc [int, bool] bool proc [real] none proc seq ptr int proc none

II.3 - Déclarations -

En BASEL, il y a deux typesde déclarations : les déclarations de variables et les déclarations de modes. A la différence d'ALGOL 68, il n'existe pas de déclarations d'opérateurs puisqu'il n'existe pas d'opérateurs au sens d'ALGOL 68. La structure d'une déclaration peut se schématiser de la façon suivante :



[1] - déclaration simple de variable.

[2] - déclaration conditionnelle.

[3] - déclaration de signification.

[4] - déclaration générique.

II.3.1 - Déclaration de variables -

II.3.1.1 - Déclaration simple de variable -

Elle sert à définir une variable classique, en indiquant son nom et le mode des valeurs qu'elle peut prendre [WI, 7.4].

Exemples :

a) - is is int

b) - p is proc [real] ptr real

II.3.1.2 - Déclaration conditionnelle -

Une variable déclarée de mode union dans un certain bloc peut faire l'objet d'une déclaration plus restrictive, dite déclaration conditionnelle, dans un bloc intérieur. Il y a restriction en ce sens que dans le bloc intérieur, on n'accepte pour cette variable qu'un des modes de l'union précédente.

A l'exécution, le bloc intérieur ne sera élaboré que si la variable a une valeur et que cette valeur est du mode spécifié dans la déclaration conditionnelle. Dans le cas contraire, le bloc intérieur ne sera pas exécuté.

(On remarque un certain rapport avec la notion de "relation de conformité" [WI, 8. 3. 2]).

Exemple :

```
(u is union [int, bool] ...
      (u mustbe int ...) ...)
```

Si la valeur de u n'est pas entière, le bloc intérieur n'est pas exécuté. Si elle est entière, u sera utilisée comme si elle avait été déclarée u is int dans le bloc englobant.

Remarque : l'affectation i = u (u mustbe int ...) n'est pas permise, car le bloc peut ne pas être exécuté, donc est considéré comme n'ayant pas de valeur.

II.3.1.3 - Déclaration de signification - Déclaration générique -

Ce sont des déclarations qui s'appliquent seulement aux procédures. La déclaration générique sert à déclarer un identificateur de procédure qui peut représenter à un moment plusieurs corps de procédure différents. Le mode de chacun des corps est spécifié au moyen de plusieurs déclarations de signification (means).

Exemple :

```
(p is generic,
 p means proc [int] bool,
 p means proc [real, int] ptr int ...
      p means proc [bool] none ...) ...)
```

Dans le bloc englobant, p a deux significations, dans le bloc intérieur une troisième signification lui est ajoutée.

Appel d'une procédure générique : lors de l'appel d'une procédure générique, le choix de la signification est fait, en fonction du mode des arguments dans l'appel. Ce choix est fait au moment de la compilation. Ceci est à rapprocher de la déclaration d'opération en Algol 68 [WI, 7.5].

II.3.2 - Déclaration de mode -

La déclaration de mode sert à définir des symboles et les modes que ces symboles représentent (rep).

Exemples :

a) - entier rep int

b) - list rep tuple [ptr listelem, ptr listelem]
listelem rep union [list, atom]

II.3.3 - Portée de variables et de symboles déclarés -

La portée d'une variable ou d'un symbole déclaré est analogue à celle de variables en Algol 60, sauf dans le cas de variables globales utilisées dans les procédures (Voir Ch.1, III.5).

II.3.4 - Règles relatives aux déclarations et à la définition des symboles -

1 - Dans un bloc, il ne peut pas y avoir plus d'une déclaration de la même variable (sauf pour les génériques) ou du même symbole. Ainsi, l'exemple suivant n'est pas correct :

```
(i is int, i is ptr int, ...)
```

2) - Tous les sélecteurs d'une valeur de mode struct doivent être des identificateurs différents. Ainsi, la construction suivante est incorrecte :

```
struct [int i, ptr int i]
```

3) - Lors de l'appel d'une procédure générique, la liste de paramètres effectifs (Voir Ch.1, V.1.2.1) doit être compatible (Voir Ch.1, VII) avec une seule liste de paramètres formels (Voir Ch.1, V.1.2.1), parmi toutes les listes possibles d'après les différentes significations de la procédure. Ainsi, les exemples suivants ne sont pas corrects.

a) -
```
(p means proc [int] int, p is generic,
 p means proc [ptr int] int,
 ppp is ptr ptr int,
        ...
 p (ppp))
```

b) -
```
(b is bool, p is generic,
 p means proc [union [bool, real]] int,
 p means proc [union [bool, char]] int,
        ...
 p (b) ...)
```

4) - Définition des modes récursifs.

La déclaration d'une variable de mode m implique la réservation de mémoire dont la taille dépend de m ; exemple :

a) -
```
m rep struct [int i, ptr real p]
```

Pour toute variable déclarée de mode m on réservera une zone de mémoires pouvant recevoir un entier plus un pointeur (adresse).

- On peut définir un mode m à l'aide de lui-même ; exemple :

b) - m rep tuple [int, ptr m]

Cette définition récursive de m ne pose pas plus de problème que celle de l'exemple a) : pour toute variable déclarée de mode m on réservera une zone de mémoire pouvant recevoir un entier plus un pointeur.

- Par contre, dans l'exemple suivant, une variable déclarée de mode m impliquerait la réservation d'une zone de mémoire de taille infinie.

c) - m rep tuple [int, m]

En effet : place pour une valeur de mode m
= place pour un entier + place pour une valeur de mode m

Pour des raisons analogues, une variable déclarée de mode m impliquerait la réservation d'une mémoire de taille indéfinie si, par exemple :

d) - m rep m

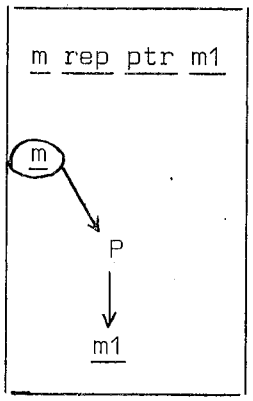
- Dans la définition d'un mode récursif, on exige donc la présence d'un ptr afin d'éliminer les problèmes qui apparaissent en c) et d).

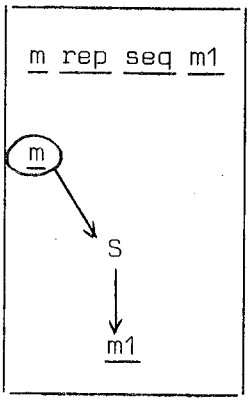
Il est évident que ce ptr ne doit pas être mis n'importe où, comme le montrent les exemples suivants :

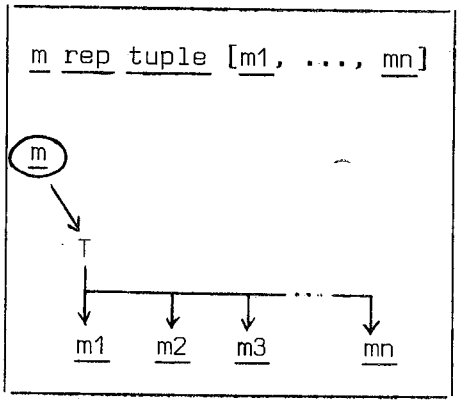
m rep union [ptr m, real] est correct

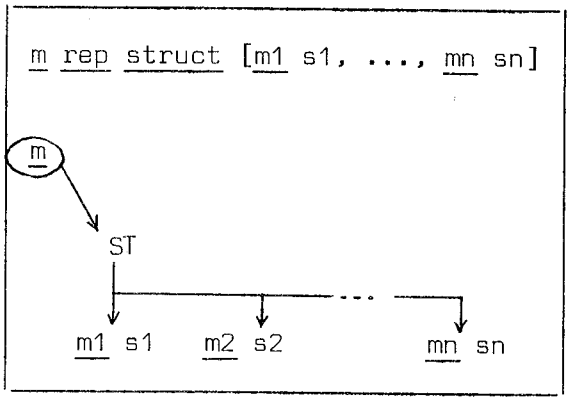
m rep union [m, ptr real] est incorrect.

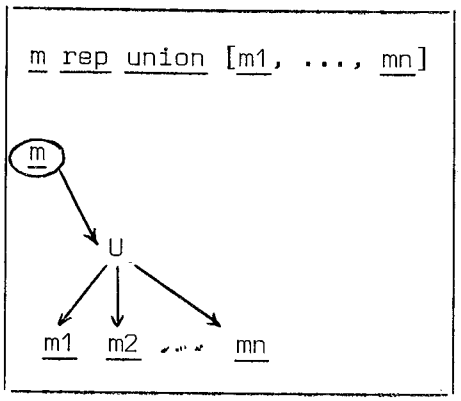
Si l'on adopte les conventions suivantes de représentation des modes, sous la forme d'arbres :

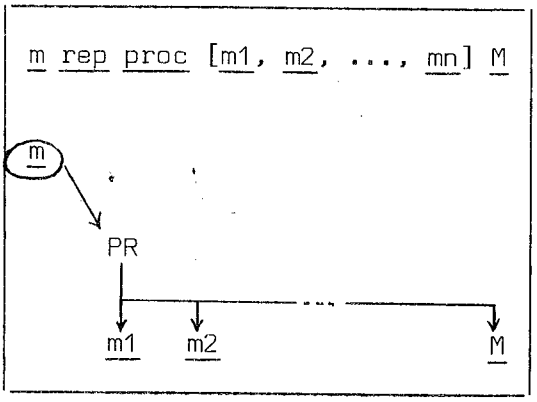


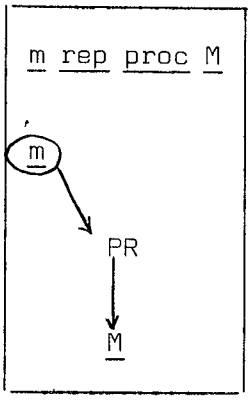










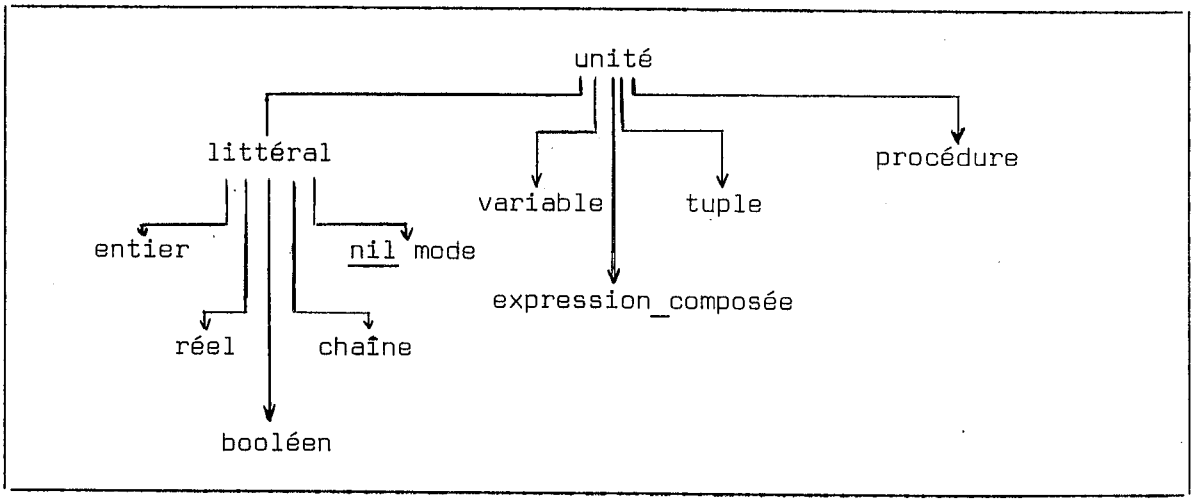


On peut dire que pour tout mode récursif m, représenté par un arbre, il doit exister un constructeur ptr (P) dans tout chemin qui lie (m) avec une occurrence de m.

Pour plus de détail, on peut se reporter au chapitre 2 (Ch.2, II.3.e en particulier).

5) - Aucun ordre n'est imposé aux déclarations d'un même bloc.

III - UNITES -



III.1 - Littéral -

Les littéraux représentent une valeur d'un certain mode.

ENTIER : suite de chiffres qui représente une valeur de mode int : 57.

REEL : deux suites de chiffres séparées par . ; représentation d'une valeur de mode real : 35.2.

BOOLEEN : les symboles true et false, de mode bool.

CHAINES : peuvent comporter un ou plusieurs caractères entre guillemets. Le mode d'une chaîne d'un seul caractère est char ; celui d'une chaîne de plusieurs caractères est tuple [char, char, ..., char], ce qui implique que l'on devra en général manipuler les chaînes comme un tout, sans accéder aux éléments. Exemples :

"a" de mode char

"FI3" de mode tuple [char, char, char].

NIL MODE : c'est la représentation d'un objet du mode spécifié, mais de valeur indéfinie. On verra une utilisation du littéral nil dans la définition de procédures récursives. (Voir Ch.1, VIII.5).
Exemple :

nil proc [int, bool] real

III.2 - Variables -

Elles correspondent à la notion classique. Elles sont identifiées par des identificateurs : a, b1, ARBRE.

III.3 - Expression composée -

Une expression composée est un ensemble ordonné de valeurs, éventuellement vide. Les valeurs qui composent l'ensemble sont celles des blocs composants de l'expression_composée ; ainsi dans :

$(bloc_1, bloc_2, \ldots, bloc_i, \ldots, bloc_n)$

la valeur de l'expression_composée est :

$\{v_1, v_2, \ldots, v_i, \ldots, v_n\}$ où v_i = valeur du $bloc_i$.

et son mode :

tuple [m1, m2, ..., mi, ... mn] où mi = mode du $bloc_i$

Si mi = none pour un i :

la valeur de l'expression_composée est :

$\{v_1, v_2, \ldots, v_{i-1}, v_{i+1}, v_n\}$,

et son mode :

tuple [m1, m2, ..., m(i-1), m(i+1), ..., mn]

Comme on l'a déjà dit dans Ch.1, I.2.3, l'élaboration d'une expression_composée est l'élaboration en série de chacun de ses blocs composants :

$bloc_1, bloc_2, \ldots, bloc_n$.

Exemples :

a) - (1) : valeur 1 et mode int

b) - ( ) : pas de valeur et mode none

c) - (3, 1.5, true, ( )) : la valeur c'est l'ensemble ordonné {3, 1.5, true} et le mode est tuple [int, real, bool].

d) - (j is int in j = 3 ; alloc j, r is real in r = read (1,2,4)) le mode est tuple [ptr int, real] (Voir Ch.1, V.2).

III.3.1 - Bloc -

Dans Ch.1, I.2.3, on a présenté la structure d'un bloc et on a décrit son élaboration.

On donne ici quelques exemples :

a) - (i is int, j is int in i = 2 ; j = i)
la valeur est la valeur de j = i ; c'est-à-dire 2, et le mode : int.

b) - (i is int, b is bool in ... ; b → 4|3.1)
la valeur est 4 ou 3.1 ; le mode est union [int, real].

c) - (u is union [int, bool] ...
(u mustbe int u = 5) ...)
le mode du bloc intérieur est none.

d) - (i is int, b is bool in ... b → i)
la valeur peut être celle de i, ou aucune : le mode est none.

III.4 - Tuple -

Tout ce qui a été dit à propos des expressions composées reste valable pour les tuples, avec les différences suivantes :

- l'ordre d'élaboration des blocs composants d'une tuple est indéfini. (Collatéral d'Algol 68 [WI, 2.2.5 b]).
- le mode de [bloc] est tuple [mode du bloc] et non mode du bloc.
- il n'existe pas la possibilité [ ].

Exemples :

a) - [1] : mode tuple[int]

b) - ["ab", ( ), 5] mode tuple [tuple [char, char], int] ;
la valeur est l'ensemble ordonné {"ab", 5}.

D'après les différences existant entre l'élaboration d'une expression_composée et celle d'une tuple, on remarque que l'élaboration de :

a = (b, b = a ; ( )) conduit à intervertir les valeurs des variables a et b ; car l'élaboration de b précède celle de b = a ; ( ).

III.5 - Procédures -

Une procédure est constituée par une partie_instructions, analogue à celle d'un bloc, et qui peut être éventuellement précédée d'une partie_spécifications. La partie_spécifications sert à préciser les modes des paramètres de la procédure. Elle est omise dans le cas où la procédure n'a pas de paramètres. Les variables déclarées dans la partie_spécifications sont les paramètres formels de la procédure.

Le mode d'une procédure avec paramètres est proc [m1, m2, ..., mn] m, où mi est le mode de la i-ème variable déclarée dans la partie_spécifications,et m est le mode résultant de l'élaboration de la partie_instructions. Le mode d'une procédure sans paramètres est proc m.

Exemples :

a) - <p is ptr int in alloc p>
mode : proc [ptr int] ptr ptr int

b) - <x is real, y is real in x = (y, y = x ; ( )) ; ( )>
mode : proc [real, real] none

VARIABLES LOCALES A LA PROCEDURE : ce sont les variables déclarées dans un bloc intérieur à la procédure. Elles peuvent être utilisées dans les blocs où elles sont locales ou globales.

Exemples :

<v1 is int, v2 is ptr int in (r is real in ...)>
v1 et v2 sont les paramètres,
r est une variable locale.

VARIABLES GLOBALES DE LA PROCEDURE : ce sont les variables déclarées dans un bloc qui englobe la procédure, et qui ne sont pas déclarées dans un bloc intérieur. Lors de l'élaboration de la procédure, la valeur de la variable globale peut être utilisée ; mais il est interdit, en BASEL, de changer la valeur d'une variable globale d'une procédure ; c'est-à-dire, il est interdit de faire des affectations. Lors de l'appel de la procédure, ce sont les valeurs des variables globales au moment de la définition de la procédure qui sont utilisées et non les valeurs au moment de l'élaboration de l'appel. Par exemple :

```
(a is int in a = 3 ;
     (p is proc int in p = <a> ;
      p ; a = 2 ; p))
```

l'appel de p est toujours identique à un appel de <3>.

Une procédure peut être affectée à une variable déclarée de mode proc ... si les modes sont les mêmes. La portée d'une variable procédure est celle de sa déclaration et elle n'est pas fonction des variables globales utilisées dans la procédure. L'utilisation des variables globales d'une procédure est absolument différente de celle d'Algol 68 [WI,2.2.4.2b]. Les exemples présentés dans Ch.1, VIII.6 montrent cette différence.

IV - SEQUENCES -

Une séquence [WI, 2.2.3.3] est un ensemble ordonné de n valeurs, toutes de même mode m, appelées éléments. Le mode de la séquence est seq m.

n est la longueur de la séquence ; si n = 0 la séquence est indéfinie et elle n'a pas d'éléments ; si n > 0 la séquence est définie.

Les éléments d'une séquence sont toujours rangés dans la mémoire libre, de façon consécutive : le i-ème élément suit le (i-1) ème.

On a accès aux éléments d'une séquence au moyen d'indices, le premier élément a toujours l'indice 1.

Une séquence est déterminée par :

1) - sa longueur

2) - l'adresse ou emplacement du premier élément

3) - le mode de ses éléments.

BASEL permet de réaliser certaines opérations sur les séquences que l'on exposera dans (Ch. 1, V.1.2.2, choix des éléments) ; (Ch.1, V.3, génération) ; (Ch.1, V.2, longueur).

V - LES EXPRESSIONS -

Les expressions en BASEL interviennent dans la construction des instructions. L'élaboration d'une expression donne en général une valeur d'un certain mode ; dans le cas où il n'y a pas de valeur le mode est none. Les expressions peuvent se combiner entre elles d'après les règles de la syntaxe en donnant des expressions nouvelles. Il existe cinq catégories d'expressions :

expressions	- opérandes - opérations - séquences et boucles - choix d'un élément dans une valeur structurée - affectations

V.1 - Opérandes -

Ils se présentent comme une valeur ou comme une suite de plusieurs valeurs. La valeur ou la suite de valeurs peuvent être précédées de modes, comme on le verra dans Ch.1, VII.3.9.

V.1.1 - Opérande sous la forme d'une valeur -

L'opérande est une unité ; sa valeur et son mode sont ceux de l'unité (Voir Ch.1, III).

Exemples :

a) - 3

b) - (4, 2.8, false)

c) - <r is real, j is int in expo (mul (j, ln r))>

V.1.2 - Opérande sous la forme de suite de valeurs -

Une suite valeur valeur est élaborée en fonction des modes des deux valeurs. Le résultat de cette élaboration est une nouvelle valeur, qui peut, à son tour être suivie d'une autre valeur, et ainsi de suite.
Le mode de la première valeur dans une suite de deux valeurs fait distinguer deux cas :

- les appels de procédures avec paramètres,
- les choix d'éléments des séquences.

V.1.2.1 - Appels de procédures avec paramètres -

Le mode de la première valeur doit être proc [m1, m2, ..., mn] m, avec n ⩾ 1 ; il est possible de faire des modifications pour obtenir ce mode (Voir Ch.1, VII.3.1.a). Il s'agit d'une procédure avec n paramètres de modes m1, m2, ... et mn respectivement, qui donne un résultat de mode m. La deuxième valeur représente le(s) paramètre(s) effectif(s) pour appeler la procédure. Les modes des paramètres effectifs (qui peuvent être modifiés, voir Ch.1, VII.3.5) doivent être ceux des paramètres formels spécifiés dans la déclaration de la procédure (Voir Ch.1, II.2). Dans le cas où il s'agit d'une procédure générique (Voir Ch.1, II.3.4-3) les modes des paramètres effectifs doivent coïncider avec les modes d'une seule liste de paramètres formels parmi toutes les listes possibles d'après les différentes significations de la procédure (Voir Ch.1, II.3.1.3),cela pour que la détermination de la signification ne soit pas ambiguë.

L'élaboration de l'appel implique l'élaboration du corps de la procédure en faisant l'identification des paramètres formels aux paramètres effectifs. Comme résultat de l'élaboration de l'appel, il y aura une valeur de mode m, ou parfois absence de valeur si m = none.

Exemples :

a) - p is proc [int, int] bool,
a is int, b is int

.
.
.

p = <i is int, j is int in grt (i, j)>

.
.
.

p(a, b)

.
.
.

l'appel p(a, b) produit l'élaboration de :

(i is int, j is int in i = a ; j = b ; grt (i, j))

b) - p is proc [int] ...

Les appels qui suivent sont corrects :

- p (3) car le mode de (3) est int.
- p 3 car le mode de 3 est int (le blanc est significatif ici).
- p((3)) car le mode de ((3)) est int.
- p plus (i, j) car le mode de plus (i, j) est int (voir Ch.1,V.2)

Mais l'appel suivant n'est pas correct :

- p [3] car le mode de [3] est tuple [int].

c) - p is proc [int, bool] ...

Les appels qui suivent sont corrects :

- p (3, true) car le mode de (3, true) est tuple [int, bool] ce qui implique deux valeurs, la première de mode int et la deuxième de mode bool.
- p [3, true]
- p ([3, true])
- p [plus (i, j), eq (k, l)]
- p ((3, true))
- si t is tuple [int, bool] : p t

Mais l'appel qui suit est incorrect :

- p [(3, true)] car le mode de [(3, true)] est tuple [tuple [int, bool]],

d) - p is proc [int] proc [real] bool

.
.
.

p 3 2.5 ,

l'élaboration de p 3 2.5 consiste à élaborer p 3 d'abord,ce qui donne comme résultat une valeur de mode proc [real] bool ; l'élaboration de cette valeur suivie de 2.5 donne comme résultat une valeur de mode bool.

On verra par la suite d'autres exemples de procédures (voir Ch.1, VIII), en particulier l'utilisation de procédures récursives (voir Ch.1, VIII.5).

V.1.2.2 - Choix d'éléments de séquences -

Le mode de la première valeur doit être seq m (il est possible de faire des modifications pour l'obtenir, voir Ch.1, VII.3.1).
Celui de la deuxième valeur représente l'indication pour choisir soit un élément de la séquence, soit une sous-séquence de la séquence originale.

V.1.2.2.1 - Choix d'un élément -

La deuxième valeur est appelée indice, son mode doit être int (il est possible de faire des modifications, voir Ch.1, VII.3.6).
Sa valeur indique quel élément de la séquence doit être choisi. L'élaboration donne comme résultat une valeur de mode ptr m, qui est l'adresse où se trouve l'élément choisi de la séquence de mode seq m.

Exemple :

```
... s is seq real, ... p is proc [int] int, ....
                    .
                    .
                    .
p = <i is int, .... ; 4>
```

l'élaboration de s(4), ou de s 4, ou de s (plus (2,2)) ou de s p(3), donne comme résultat l'adresse du quatrième élément de la séquence s. Le mode de cette adresse est ptr real.

V.1.2.2.2 - Choix d'une sous-séquence -

La deuxième valeur doit être de mode tuple [int, int](il est possible de faire des modifications, voir Ch.1, VII.3.6.), c'est-à-dire qu'il s'agit d'un ensemble de deux valeurs de mode int, appelées indice inférieur et indice supérieur, respectivement. L'élaboration donne comme résultat une séquence de mode seq m dont :

le premier élément est l'élément d'ordre égal à l'indice inférieur de la séquence originale,
le deuxième élément est l'élément d'ordre égal à l'indice inférieur + 1 de la séquence originale, etc. ;
le dernier élément est l'élément d'ordre égal à l'indice supérieur de la séquence originale.

L'indice supérieur doit être plus grand ou égal à l'indice inférieur. La longueur de la nouvelle séquence est la différence entre les indices supérieur et inférieur, plus 1.

Exemple :

```
... s is seq real, p is proc [int] int
                ....
    p = <..... ; 4>
```

l'élaboration de :

s[4, 10], ou de s(4, 10) ou de s(plus (2,2), plus (p(1), 6)) donne comme résultat une séquence de 7 éléments, dont les valeurs sont les mêmes que celles des éléments numéros 4 à 10 de s.

V.1.3 - Catégories d'opérandes -

Dans ce qui suit, on qualifiera de "simples", "doubles" ou "triples" des opérandes ayant respectivement les formes :

- (opérande) ou opérande,
- (opérande, opérande) ou [opérande, opérande],
- (opérande, opérande, opérande) ou [opérande, opérande, opérande], que l'on représentera par v1, (v1, v2) et (v1, v2, v3).

V.2 - Opérations -

Une opération est l'application d'un opérateur à un opérande, ce qui donne toujours une valeur d'un certain mode. A chaque opérateur, on associe une élaboration particulière et on fixe le mode de l'opérande qui doit l'accompagner. Tous les opérandes sont susceptibles de modification (voir Ch.1, VII et Ch.1, VII.3.1) afin que l'opération soit possible.

V.2.1 - Description des opérateurs -

V.2.1.1 - Opérateurs sur pointeurs -

1 - alloc : l'opérande est "simple", de mode m différent de none. L'opération consiste en la recherche d'un emplacement libre dans la mémoire libre, où l'on range la valeur de l'opérande. Le mode du résultat est ptr m.

Exemple : alloc 2.

2 - set : l'opérande est "double" : (v1, v2) de mode tuple [ptr m, m]. v2 est rangé dans la mémoire libre dans l'adresse repérée par v1. Le mode du résultat est **ptr** m.

Exemple : set (p, 5) avec p is ptr int.

Remarque : set sert en particulier à faire des affectations à des éléments de séquences.

3 - same : l'opérande est "double" (v1, v2) de mode tuple [ptr m, **ptr** m]. Le résultat, de mode bool est égal à true si v1 est égal à v2, et false dans le cas contraire.

Exemple : same (p, q) avec p et q de mode ptr real.

V.2.1.2 - Opérateurs sur des séquences -

4 - length : l'opérande est "simple" : v1, de mode seq m. Le résultat est la longueur de la séquence, de mode int.

Exemple : i = length s.

5 - change : l'opérande est double (v1, v2), de mode tuple [seq m, int]. La longueur de v1 est incrémentée de v2. Si v2 est positive, on ajoute v2 éléments de valeur nil m à la fin de la séquence. Dans le cas où v2 est négative, on enlève les v2 éléments de la fin de la séquence v1.

Exemple : change (s, 5).

V.2.1.3 - **Opérateurs booléens, de comparaison, arithmétiques, de conversion et d'entrée-sortie : ils réalisent les opérations classiques.** Ils sont détaillés dans l'Appendice II.

V.3 - Séquences et boucles -

Une boucle se présente en BASEL sous la forme d'expression times expression.

L'expression qui précéde times doit être de mode int, après modification, si nécessaire (Voir Ch.1, VII.3.2). Sa valeur indique combien de fois doit s'élaborer l'expression qui suit times ; si elle est négative ou nulle, l'expression qui suit times n'est pas élaborée. Si l'expression qui suit times est de mode m différent de none, l'élaboration de la boucle implique la construction d'une séquence de mode seq m, dont les éléments sont les résultats successifs de l'élaboration qui se répète.

L'élaboration d'une boucle donne donc, comme résultat :

- soit une séquence de mode seq m,
- soit aucune valeur, mode none.

Ainsi dans les exemples a), b) et c) ci-dessous, il y a construction des séquences ; mais pas dans d), e) et f).

a) - 10 times 1

La valeur 1 est élaborée 10 fois, en même temps on construit une séquence de mode seq int, dont les 10 éléments ont tous la valeur 1.

b) - 5 times [3 ; 2.5]

la valeur 3 et la valeur 2.5 sont élaborées 5 fois ; à chaque élaboration la valeur 2.5 est celle de toute l'expression. On construit une séquence de 5 éléments tous de mode tuple [real] et de valeur 2.5.

c) - i = 20 ; 4 times ((1) ; i = plus (i, 1))

l'expression (1) est élaborée 4 fois ; chaque fois sa valeur est celle de i = plus (i, 1) ; l'affectation i = plus (i, 1) (voir Ch.1, V.2 et V.5) permet de faire prendre à i une nouvelle valeur chaque fois. Une séquence est construite avec les valeurs de i. La séquence a donc 4 éléments de mode int et de valeur 21, 22, 23, 24 respectivment.

d) - i = 20 ; 4 times ((1) ; i = plus (i, 1) ; ( ))

l'expression (1) n'a pas de valeur, son mode est none. Alors, l'élaboration de la boucle est différente de celle de l'exemple c), ci-dessus, **par le fait** que la séquence n'est pas construite. Le mode de la boucle est none.

e) - Procédure qui fait la somme de deux matrices (a x.b).

```
som_mat is proc [seq seq real, seq seq real, int, int] seq seq real,
som_mat = <ma is seq seq real, mb is seq seq real,
           a is int, b is int
           in
(i is int, j is int in
 i = 1 ; a times
   (j = 1 ; b times (rwrite(2,plus (ma i j, mb i j)) ;
                     j = plus (j, 1) ; ( )) ;
    i = plus (i, 1) ; ( ))>
```

f) - Procédure qui fait la somme des éléments d'une séquence de mode seq int.

```
sommeseq is proc [seq int] int
sommeseq = <s is seq int in
            (n is int, i is int in n = 0 ; i = 0 ;
             length s times (n = plus (n, s (i = plus (i, 1))) ;
            ( )) ; n)>
```

Remarque : le mode de s (i = plus (i, 1)) est ptr int ; il y aura donc une modification (dérepérage) pour obtenir une valeur de mode int (Voir Ch.1, VII.2.1).

V.4 - Choix d'un élément d'une valeur structurée -

Dans sélecteur of expression, l'expression doit être de mode struct [..., m s,...] ou ptr struct [..., m s,...](après modification si nécessaire, voir Ch.1, VII.3.3), où s doit être l'identificateur qui précéde le symbole of. Le champ identifié par s est choisi ; le mode de la valeur résultante est m (si l'expression est de mode struct ...) ou ptr m (si l'expression est de mode ptr struct ...).

Exemples :

a) z is struct [real a, real b]
   a of z est une valeur real, la première des deux valeurs qui **composent z.**

b) p is proc [int] struct [real x, real j],
   k is int
   x of p(k) est la valeur du premier champ du résultat de l'appel p(k).

V.5 - Affectations -

Une affectation consiste en deux parties : la partie gauche et la partie droite, séparées par le symbole =. La partie gauche est toujours un nom, la partie droite, une expression ; le **mode de** la partie droite doit être compatible avec celui de la partie gauche (Voir Ch.1, VII).

Un nom est un identificateur de variable ou un sélecteur qui identifie un champ d'une variable de mode struct [...]. Un nom représente l'emplacement où est rangée la valeur de la variable ou du champ.

La valeur de la partie droite d'une affectation est rangée dans l'emplacement déterminé par la partie gauche.

Il peut y avoir des affectations multiples, de la façon classique, car une affectation est considérée comme une expression ayant la valeur de sa partie droite.

Exemples :

a) - i = j

b) - a of z of w = r

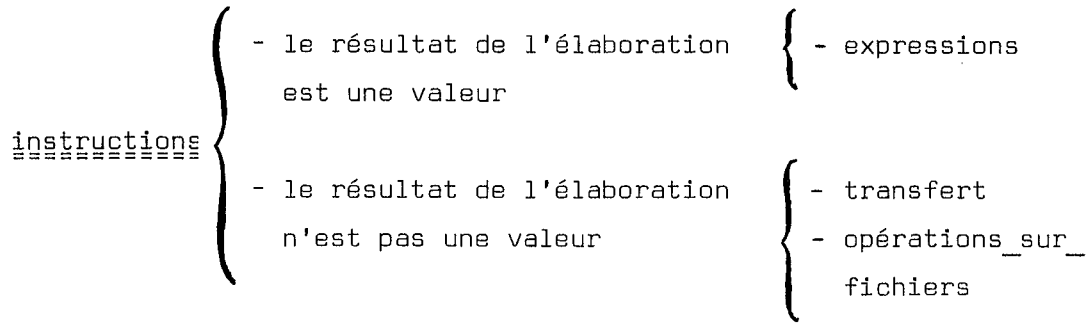
c) - p is proc [int] real,
q is proc [int] real,
.
.
.
p = <...> ; ... q = p ;

d) - s1 is seq int, i is int in i = 0 ; ...
l'affectation :
s1 = 10 times (i = plus (i, 1)) ;
est correcte ;
mais l'affectation :
s1 = 10 times (i = plus (i, 1) ; ( )) ;
ne l'est pas.

e) - l'affectation suivante est incorrecte :
(i is int, ...
i = (u mustbe bool in ... ; 3) ...)
car l'expression (u mustbe bool in ... ; 3) peut ne pas avoir de valeur, et son mode est donc none.

f) - affectations multiples :
a = b = c of d = mul (i, (j is int in ... ; k))

VI - LES INSTRUCTIONS -



Toute instruction peut être précédée par une ou plusieurs étiquettes.

VI.1 - Etiquettes -

Ce sont des identificateurs suivis du symbole : . Le traitement des étiquettes est classique. La portée des étiquettes se détermine donc comme celle des identificateurs.

Exemple :

```
L1 : L2 : a = b
```

VI.2 - Instructions qui ne donnent pas de valeur -

VI.2.1 - Transfert -

L'instruction de transfert est le symbole goto suivi d'un identificateur d'étiquette.

Comme on l'a dit au Ch.1, I.2.3, la rencontre d'une instruction de transfert avec une étiquette extérieure au bloc où se trouve l'instruction de transfert termine l'élaboration du bloc. On dit que l'élaboration n'est pas complétée.

Un transfert intérieur au bloc fait continuer l'élaboration à partir du point identifié par l'étiquette.

Restriction relative à l'instruction de transfert à l'intérieur du corps d'une procédure : il est interdit dans une procédure BASEL, d'écrire des instructions de transfert, telles que l'identificateur qui suit le symbole goto soit global à la procédure.

Exemples :

a) - (... (... goto L ; ... ; 3) ... L : ...)
goto L termine l'élaboration du bloc intérieur qui n'a pas de valeur.

b) - l'exemple suivant est incorrect :
(..., p is proc [bool] ptr bool, ...
in
p = <b is bool in ... goto L ; ...> (on suppose qu'il n'existe pas L : à l'intérieur de <...>).
...
L : ... ; p (true) ; ...)

VI.2.2 - Opérations sur fichiers -

L'opérande qui accompagne input, output ou close doit être "triple". La première valeur représente une unité d'entrée-sortie ; la deuxième est une chaîne de caractères qui représente le nom d'un fichier ; la troisième est une procédure sans paramètres et sans résultat qui est appelée en cas de fin de fichier. Le mode de l'opérande est donc :
tuple [int, tuple [char, ..., char], proc none].
input spécifie l'ouverture d'un fichier d'entrée,
output celle d'un fichier de sortie et
close spécifie la fermeture d'un fichier.

Exemples :

a) - input [2, "données", p1]

b) - output [k, "sortie", p2]

c) - close (3, fichier 1, proc_fermeture).

VII - MODIFICATION -

La modification est un processus qui permet de transformer une valeur de mode n en une valeur de mode m, si n est compatible avec m, lorsque le contexte impose une valeur de mode m et qu'on a une valeur de mode n.

VII.1 - Compatibilité de modes -

Le mode n est compatible avec le mode m, et on le note : $n \mathcal{C} m$, si et seulement si :

1 - n = m ; ou

2 - n = ptr n1 et $n1 \mathcal{C} m$; ou

3 - n = proc n1 et $n1 \mathcal{C} m$; ou

4 - n = tuple [n1, n2, ..., np] ou
n = struct [n1 s1, n2 s2, ..., np sp], et
m = tuple [m1, m2, ..., mp], et $ni \mathcal{C} mi$, pour tout $i \in [1, p]$; ou

5 - n = tuple [n1, n2, ..., np] ou
n = struct [n1 s1, n2 s2, ..., np sp], et
m = struct [m1 s1, m2 s2, ..., mp sp], et $ni \mathcal{C} mi$ pour tout $i \in [1, p]$; ou

6 - n = tuple [n1, n2, ..., np] ou
n = struct [n1 s1, n2 s2, ..., np sp] et
m = seq m1 et $ni \mathcal{C} m1$ pour tout $i \in [1, p]$; ou

7 - m = union [m1, ..., mp] et $n \neq$ union [n1, ..., nq] et
il existe i tel que mi = n ; ou

8 - m = union [m1, ..., mp] et n = union [n1, ..., nq] et
pour tout $j \in [1, q]$ il existe $i \in [1, p]$ tel que mi = nj

9) - n ∉ ptr ... ptr int ; ou
n ∉ ptr ... ptr real ; ou
n ∉ ptr ... ptr bool ; ou
n ∉ ptr ... ptr char ; ou
n ∉ ptr ... ptr seq n1 ; ou
n ∉ ptr ... ptr tuple [n1, n2, ..., np] ; ou
n ∉ ptr ... ptr struct [n1 s1, n2 s2, ..., np sp] ; ou
n ∉ ptr ... ptr union [n1, n2, ..., np] ; ou
n ∉ ptr ... ptr proc [n1, n2, ... np] N

et m = none.

n n'est pas compatible avec m est noté n ∉ m.

VII.2 - Les modifications -

Il existe en BASEL sept types de modifications. Elles peuvent être combinées entre elles.

Les modifications possibles sont :

- dérepérage [WI, 8.2.1]
- appel [WI, 8.2.2]
- extension [WI, 8.2.4]
- séquentialisation
- structuration
- destructuration
- suppression [WI, 8.2.8]

Les modifications "dérepérage", "appel", "extension" et "suppression", peuvent correspondre aux "coercions" "dereferencing", "deproceduring", "uniting" et "voiding" d'ALGOL 68, comme on l'a signalé ci-dessus. Il n'existe pas en BASEL de correspondant de "proceduring" [WI, 8.2.3], widening [WI, 8.2.5], rowing [WI, 8.2.6] et "hipping" [WI, 5.2.7]. Séquentialisation, structuration et déstructuration, n'ont pas de correspondant en ALGOL 68.

VII.2.1 - Dérepérage -

Une valeur est dérepérée quand elle est de mode ptr m et que l'on doit la transformer en une valeur de mode m.

L'élaboration du dérepérage consiste à obtenir l'objet repéré par la valeur initiale, c'est-à-dire, l'objet qui se trouve dans l'adresse qui est la valeur initiale.

Exemple :

```
i is int,
p is ptr int,
     .
     .
     .
i = p ; la valeur de p est dérepéré, c'est l'entier qui se
        trouve dans l'adresse valeur de p, qui est affecté
        à i.
```

VII.2.2 - Appel -

On élabore un appel lorsqu'il est nécessaire d'avoir une valeur de mode m et que l'on a une valeur de mode proc m.

L'élaboration consiste à appeler la procédure de mode proc m, il s'agit toujours d'une procédure sans paramètres.

Exemple :

```
a is real,
p is proc real,
     .
     .
     .
a = p ; la valeur real, résultat de l'élaboration de l'appel
        de p est affectée à a.
```

VII.2.3 - Extension -

Une valeur est étendue quand son mode est n (n ≠ union [...]) et qu'il est nécessaire d'avoir une valeur de mode union [m1, ..., mp, n] ou quand son mode est union [n1, ..., nq] et qu'il est nécessaire d'avoir une valeur de mode union [m1, ..., mp, n1, ..., nq].

L'élaboration de l'extension consiste à construire une nouvelle valeur à partir de la valeur initiale et de son mode.

Exemple :

```
u is union [int, bool],
      .
      .
      .
u = 3 ; la valeur 3 de mode int est modifiée dans une valeur
        de mode union [int, bool] formée par 3 et l'indication
        de mode int.
```

VII.2.4 - Séquentialisation -

La séquentialisation d'une valeur est faite, lorsque cette valeur est de mode tuple [m, m, ..., m] ou struct [m s1, ..., m sp] et qu'il est nécessaire d'avoir une valeur de mode seq m.

L'élaboration de la séquentialisation consiste à construire une séquence dont les éléments sont les valeurs de la tuple ou de la structure, en conservant leur ordre initial.

Exemples :

```
s is seq int
z is struct (int a, int b, int c) ;
a of z = 1 ; b of z = 10 ; c of z = 100;
```

a) - s = z ; s a 3 éléments : 1, 10, 100

b) - s = [0, 5, 10, 15]; s a 4 éléments : 0, 5, 10, 15

VII.2.5 - Structuration -

Une valeur de mode tuple [m1, ..., mp] est structurée si elle est transformée en une valeur de mode struct [m1 s1, ..., mp sp].

L'élaboration de la structuration consiste à identifier chacun des éléments composants de la tuple avec un sélecteur.

Exemples :

```
t is tuple [int, real],
z is struct [int i, real r],
      .
      .
      .
```

z = t : la valeur de t structurée devient de mode struct [int i, real r].

z = (1, 2.1) la valeur de l'expression composée (1, 2.1) de mode tuple [int, real] ; devient de mode struct [int i, real r].

VII.2.6 - Destructuration -

Une valeur est déstructurée si elle est de mode struct [m1 s1, ... mp sp] et il est nécessaire d'avoir une valeur de mode tuple [m1, ..., mp].

L'élaboration de la déstructuration fait un tout d'un ensemble structuré en champs.

Exemple :

```
t is tuple [int, real].
z is struct [int i, real r],
      .
      .
      .
```

t = z : l'ensemble de mode tuple [int, real] et non l'ensemble de mode struct [int i, real r] est affecté à t.

VII.2.7 - Suppression -

Une valeur est supprimée quand elle doit être ignorée (voir Ch.1, VII.3). L'élaboration de la suppression implique l'élaboration d'un appel si la valeur à supprimer est de mode proc m ou compatible avec proc m. L'appel se fait pour élaborer les effets de bord de la valeur à supprimer. On peut dire que le mode de la valeur à supprimer doit être compatible avec none.

```
p is proc proc ptr proc int,
      .
      .
      .
p ; appel de p : valeur résultante, proc ptr proc int
      appel de la dernière valeur : valeur résultant, ptr proc int
        dérepérage de la dernière valeur : valeur résultant,
                                                proc int
          appel de la dernière valeur : valeur résultant int
            suppression de la valeur de mode int
```

VII.3 - Application des modifications -

Les modifications sont appliquées d'après le mode imposé par le contexte.

Si on a une valeur de mode n et s'il est nécessaire d'avoir une valeur de mode m, si $n \not\in m$, alors le programme est incorrect.

Il est possible de faire des modifications dans les cas suivants :

VII.3.1 - Opérande -

a) - Un opérande peut être modifié pour obtenir une valeur de mode proc [m1, ..., mn] m ou seq m (voir Ch.1, V.1.2.1 et Ch.1, V.1.2.2)

b) - Un opérande peut être modifié pour obtenir une valeur du mode imposé par un opérateur déterminé (voir Ch.1, VI.2.2).

VII.3.2 - Boucles -

La valeur de l'expression qui précéde le symbole times peut être modifiée pour obtenir une valeur de mode int, (voir Ch.1, VI.3).

VII.3.3 - Choix d'un champ -

La valeur de l'expression qui suit sélecteur of peut être modifiée pour obtenir une valeur de mode struct [..., m sélecteur, ...] ou ptr struct [..., m sélecteur, ...].

VII.3.4 - Affectation -

L'expression de la partie droite d'une affectation peut être modifiée afin d'obtenir une valeur de mode égal à celui de la partie gauche (voir Ch.1, V.5.2).

VII.3.5 - Appel de procédures avec paramètres -

Chacun des paramètres de la liste des paramètres effectifs peut être modifié lors de l'appel d'une procédure avec paramètres, afin d'obtenir des valeurs de même mode que les paramètres formels (voir Ch.1, V.1.2.1 et Ch.1, II.3.4.3).

VII.3.6 - Choix d'éléments d'une séquence -

La valeur qui suit une valeur de mode seq m peut être modifiée afin d'obtenir une valeur de mode int ou tuple [int, int] (voir Ch.1, V.1.2.2).

VII.3.7 - Le symbole -> -

La valeur de l'expression qui précéde le symbole -> peut être modifiée pour obtenir une valeur de mode bool (voir Ch.1, I.2.3).

VII.3.8 - Le symbole ; -

Le symbole ; implique la suppression de la dernière valeur élaborée (voir Ch.1, I.2.3 et Ch.1, VII.2.7).

VII.3.9 - Modification explicite -

D'après la syntaxe une valeur peut être précédée d'un mode. Soit n le mode de la valeur et m le mode qui la précéde : si n ≠ m les modifications nécessaires pour transformer la valeur de mode n en une de mode m sont élaborées. Si n n'est pas compatible avec m le contexte est incorrect.

Exemples :

a) - p is proc ptr int

```
        .
        .
        .
    int p
```

b) - t is tuple [int, real]

```
        .
        .
        .
    r of struct [int i, real r] t
```

Cet exemple montre un moyen d'avoir accès aux éléments d'une tuple.

VIII - EXEMPLES DE PROGRAMMATION EN BASEL -

VIII.1 - Procédure pour concatener deux séquences de mode seq char -

```
concat is proc [seq char, seq char] seq char

concat = <s1 is seq char, s2 is seq char
          in
          (l1 is int, i is int in
          l1 = length s1 ; i = 0 ; change (s1, length s2) ;
          length s2 times
          (set (s1 (plus (l1, plus (i, 1))), s2 (i = plus
          (i, 1))) ; ( )) ; s1)>
```

VIII.2 - La fonction "car" de lisp -

```
liste rep tuple [ptr listelem, ptr listelem],

listelem rep union [liste, atom],

atom rep char,

car is proc [ptr listelem, ptr listelem] listelem

car = <a is ptr listelem, b is ptr listelem in listelem a>
```

si l est déclarée : "l is liste", l'appel "car l" est l'application de la fonction "car" à la liste l.

Remarque : La définition et l'appel de car tels qu'ils sont programmés ci-dessus montrent comment on peut avoir accès aux éléments d'une tuple.

VIII.3 - L'instruction "case" d'Algol 68 -

```
case is proc [int, seq proc none] none

case = <i is int, s is seq proc none in s i ; ( )>
```

VIII.4 - Passage de procédures comme paramètres d'autres procédures -

```
(i is int ; j is ptr int, k is proc int,
 p is proc [int, ptr int, proc int] int
 in
 i = 1 ; j = alloc 4 ; k = <int set (j, plus (i, 1))> ;
 p = <a is int, b is ptr int,  c is proc int
      in int set (b, plus (a, c))> ;
 p (i, j, k))
```

L'appel p(i, j, k) implique l'élaboration de :

```
(a is int, b is ptr int, c is proc int
 in
 a = i ; b = j ; c = k; ← identification des paramètres
 set (b, plus (a, c)))
```

L'élaboration de la partie_instructions de p se fait d'après :

1) - élaboration de c : le mode est proc int et il faut le mode int, il faut faire la modification appel

appel c :

1.1) élaboration de plus (i, 1) : valeur 2 ; mode int

1.2) élaboration de set (j, 2) : le mode est ptr int ; il faut le mode int ; il faut faire la modification dérepérage.

dérepérage j : valeur 2, mode int

2) - élaboration de a : valeur 1, mode int

3) - élaboration de plus (a, c) : valeur 3, mode int

4) - élaboration de int set (b, plus (a, c)) : valeur = 3, mode int

On voit donc qu'avec un mécanisme uniforme de liaison des paramètres, on obtient le même effet que le passage par valeur, par référence ou par nom selon le mode de ces paramètres.

VIII.5 - Procédures récursives -

Une procédure récursive est appelée à l'intérieur de son corps, mais elle est déclarée à l'extérieur. D'après les règles d'utilisation des variables globales de procédures, l'identificateur qui identifie la procédure doit avoir une valeur au moment de la définition de la procédure, puisque c'est cette valeur qui est prise. L'écriture des procédures récursives en BASEL est faite au moyen d'un littéral nil de la façon suivante :

Exemple :

```
a is ptr proc [int, int] int,
                .
                .
                .
a = alloc nil proc [int, int] int ;

set (a, <p is int, q is int in
        eq (p, 0)—> plus (q ; 1) | eq (q, 0)
                 —> a[1] (minus (p, 1), 1) |
        a[1] (minus (p, 1), a[1] (p, minus (q - 1)))>
```

[1] : au moment de cette utilisation de a, a est défini, et repère nil. Après l'élaboration de l'opération set, la valeur repérée par a est changée ; c'est la procédure qui calcule la fonction d'Ackermann. Tout appel de a implique une modification (dérepérage).

VIII.6 - Variables globales utilisées dans une procédure -

VIII.6.1 - Exemple 1 : procédure qui calcule le nombre de fois qu'on l'a appelée :

```
a) - (k is ptr int, p is proc int in k = alloc 0 ;
      p = <set (k, plus (k, 1))> ;
      (k is int in ... p ; ... p ; ...))
```

On est obligé de travailler en deux niveaux d'imbrication et de redéfinir k pour protéger le compteur utilisé par p. Une autre façon de le faire est la suivante :

```
b) - (p is proc int in
       (k is ptr int in k = alloc 0 ;
       p = <set (k, plus (k, 1))> ) ;
       ... p ; ... p ; ...)
```

L'identificateur k est "perdu" à la sortie du bloc intérieur, mais l'emplacement de la mémoire libre repéré par la valeur k au moment de la définition de p n'est pas "perdu", et c'est lui qui est utilisé par p.

VIII.6.2 - Exemple 2 : composition des fonctions.

```
(fonc rep proc [real] real,

 compose is proc [fonc, fonc] fonc,

 fonction1 is fonc, fonction2 is fonc,
 fonction3 is fonc in
          .
          .
 fonction1 = < ... > ; fonction2 = < ... > ;

 compose = <f is fonc, g is fonc

                    in

                 <x is real in f(g (x))>> ;
          .
          .
          .
fonction3 = compose (fonction1, fonction2)[1] ;

           compose (fonction1, fonction2) (3.5)[2] ; ...)
```

L'appel compose (fonction1, fonction2) dans [1] et dans [2] donne comme résultat une valeur de mode proc [real] real. Dans [1] cette valeur est affectée à fonction3. Dans [2] elle est appelée avec le paramètre réel 3.5 ; les valeurs de f et de g qui sont alors utilisées sont celles qui ont été générées en même temps que la procédure, bien que f et g ne soient plus accessibles puisque l'appel de compose est terminé. Ces valeurs étaient globales au moment de l'élaboration de <x is real in f(g (x))>, résultat de compose.

VIII.7 - Calcul formel de la dérivée de f (x) par rapport à x.

(Traduction de l'Algol 68 [WI, 11.11]).

```
(form rep union [ptr const, ptr var, ptr triple, ptr call],

const rep struct [real value],

var rep struct [char name, real value],

triple rep struct [form left_operand, int operator, form rigth_operand],

function rep struct [ptr var bound_var, form body],

call rep struct [ptr function function_name, form parameter],

signe_plus is int,

signe_moins is int,

signe_fois is int,

signe_par is int,

signe_puis is int,

zero is ptr const, one is ptr const,

eq is generic, eq means proc [int, int] bool,
               eq means proc [real, real] bool,
               eq means proc [form, form] bool,

plus is generic, plus means proc [real, real] real,
                 plus means proc [form, form] form,

minus is generic, minus means proc [real, real] real,
                  minus means proc [form, form] form,

mul is generic, mul means proc [real, real] real,
                mul means proc [form, form] form,
```

```
div is generic, div means proc [real, real] real,
                div means proc [form, form] form,

exp is generic, exp means proc [real, int] real,
                exp means proc [form, ptr const] form,

derivate_of is ptr proc [form, ptr var] form,

value_of is ptr proc [form] real,

a is ptr var, b is ptr var, x is ptr var,

f is form, g is form

            in

signe_plus = 1 ; signe_moins = 2 ; signe_fois = 3 ; signe_par = 4 ;
signe_puis = 5 ;

zero = alloc const [0.0] ; one = alloc const [1.0] ;

eq = <a is int, b is int in eq (a, b)> ;

eq = <a is real, b is real in eq (a, b)> ;

eq = <a is form, b is form in
    (p is bool in p = false ;
        (a mustbe ptr const in p = same (a, b)) ; p)> ;

plus = <a is real, b is real in plus (a, b)> ;

plus = <a is form, b is form in
       eq (a, zero)-> b | eq (b, zero)-> a | triple (a, signe_plus, b)> ;

minus = <a is real, b is real in minus (a, b)> ;

minus = <a is form, b is form in
        eq (b, zero)-> a | triple (a, signe_moins, b)> ;
```

```
mul = <a is real, b is real in mul (a, b)> ;

mul = <a is form, b is form in
      or (eq (a, zero), eq (b, zero))→ zero|
      eq (a, one)→ b | eq (b, one)→ a|
      triple (a, signe_fois, b)> ;

div = <a is real, b is real in div (a, b)> ;

div = <a is form, b is form in
      and (eq (a, zero), not eq (b, zero))→ zero|
      eq (b, one)→ a | triple (a, signe_par, b)> ;

exp = <a is real, b is real in exp (a, b)> ;

exp = <a is form, b is ptr const in
      or (eq (a, one), eq (b, zero))→ one
      eq (b, one)→ a | triple (a, signe_puis, b)> ;

derivate_of = alloc nil proc [form, ptr var] form ;

set (dérivate_of,

 <e is form, x is ptr var in
 (p is form in
  (e mustbe ptr const in p = zero,
   e mustbe ptr var in same (e, x)→ p = one | p = zero,
   e mustbe ptr triple in
  (u is form, v is form, udash is form, vdash is form in
   u = left_operand of e ; v = rigth_operand of e ;
   udash = derivate of (u, x) ; vdash = derivate_of (v, x) ;
   eq (operator of e) signe_plus)→ p = plus (udash, udash) |
   eq (operator of e, signe_moins)→ p = minus (udash, vdash)|
   eq (operator of e, signe_fois)→ p = plus (mul (u, vdash), mul (v, udash))|
   eq (operator of e, signe_par)→ p = div (minus (udash, mul (e, udash)), v)|
   (v mustbe ptr const in
   p = mul (mul (v, exp (u, alloc const  minus (value of v, 1) )), udash))),
```

```
    e mustbe ptr call in
      (f is ptr function, g is form, j is ptr var,
       fdash is ptr function in
       f = function_name of e ; g = parameter of e ;
       y = bound_var of f ; fdash = alloc function (y, derivate_of (body
                                                          of f, y)) ;
       p = mul (call (fdash, g), derivate_of (g, x)))) ; p) > ) ;

       value_of = alloc nil proc [form] real ;

set (value_of,

 <e is form in
 (p is real in
  (e mustbe ptr const in p = value of e,
   e mustbe ptr var in p = value of e,
   e mustbe ptr triple in
  (u is real, v is real in
   u = value_of (left_operand of e) ;
   v = value_of (rigth_operand of e) ;
   eq (operator of e, signe_plus)--> p = plus (u, v)|
   eq (operator of e, signe_moins)--> p = minus (u, v)|
   eq (operator of e, signe_fois)--> p = mul (u, v)|
   eq (operator of e, signe_par)--> p = div (u, v)|
   p = exp (u, v)),
   e mustbe ptr call in
   (f is ptr function in
    f = function_name of e ;
    set (value of bound_var of f, value_of (parameter of f)) ;
    p = value_of (body of f))) ; p) > ) ;

a = alloc var ("a", rread(1, 2, 3)) ;

b = alloc var ("b", rread(1, 2, 3)) ;

c = alloc var ("c", rread(1, 2, 3)) ;
```

```
f = plus (a, div (x, plus (b, x))) ;

g = div (plus (f, one), minus (f, one)) ;

write ( 2, valeur_of (derivate_of (g, x)))
```

CHAPITRE 2

L E T R A I T E M E N T D E S M O D E S

Dans ce qui suit, on s'intéressera successivement à la représentation des modes qui est construite et utilisée par le compilateur, aux propriétés élémentaires de ces représentations, au problème de l'équivalence des modes et à celui de la réduction des représentations.

I - REPRESENTATIONS ET NOTATIONS -

Deux cas sont à considérer : le cas simple des modes de base et le cas général des modes construits.

I.1 - Cas des modes de base -

On représentera mb l'un quelconque des modes de base et l'absence de mode :

int , real , bool , char , none

I.2 - Cas des modes construits -

Si c est un constructeur, et m1, m2, ..., mn sont des modes, un mode construit m est défini et noté par :

m = c(m1, m2, ..., mn)

On distingue alors deux possibilités, qui se traduisent par une structure différente des représentations : celle où c est l'un des constructeurs ptr, rep, tuple, struct ou proc et celle où c est le constructeur union.

I.2.1 - c ∈ {ptr, seq, tuple, struct, proc}

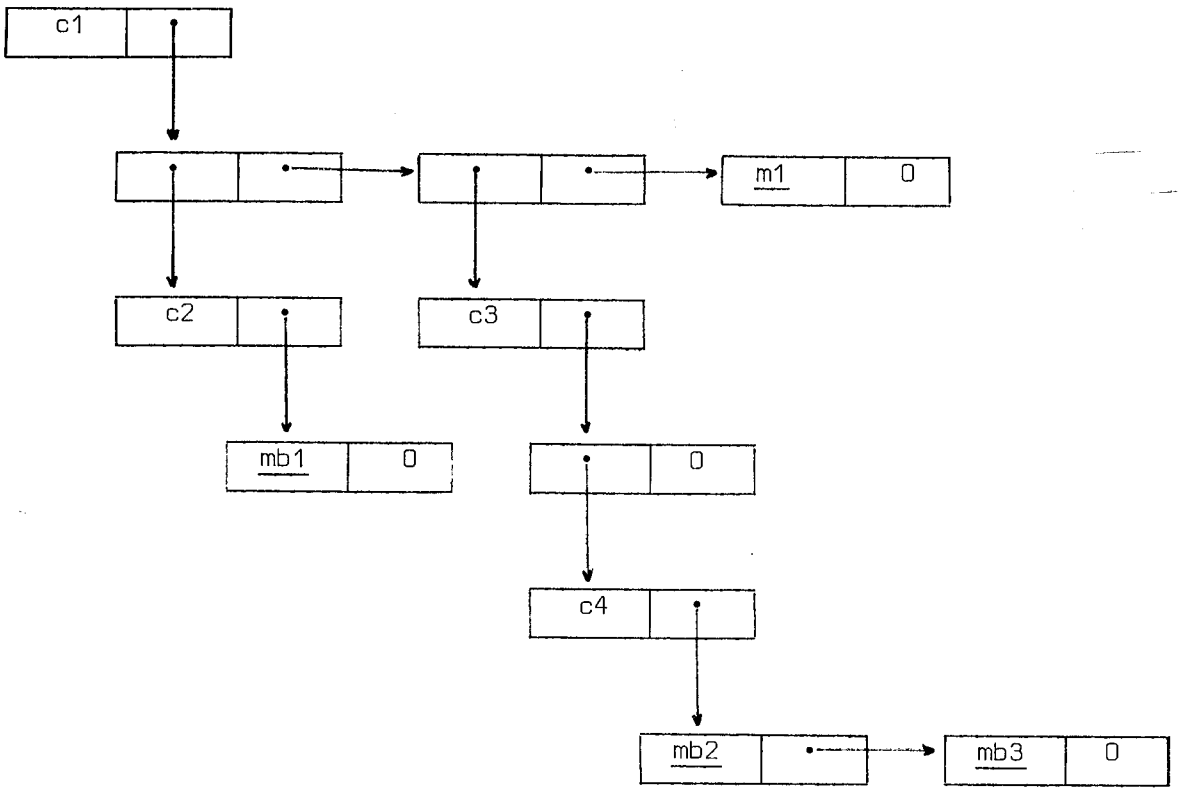
Dans ce cas, la représentation du mode a la forme d'une liste binaire dont la structure peut être formalisée par les règles suivantes ; où le non-terminal ml est choisi pour signifier "mode-liste" :

ml → (constructeur, liste)
liste → (composant, suivant)
composant → ml | mb | md | mu
suivant → fin | liste
fin → 0
constructeur → ptr | seq | tuple | struct | proc
md → un symbole qui représente un mode. Ce symbole sera associé à un mode (de type ml, mb, mu ou même encore md) au moyen d'une table appelée M-D (Mode-Déclaré). Chaque entrée dans cette table contiendra le symbole et un accès à la représentation du mode qu'il représente.
mu → mode construit avec c = union (voir Ch.2, I.1.2.2).

Par exemple, si c1, c2, c3 et c4 sont des constructeurs, si mb1, mb2 et mb3 sont des modes de base et si m1 est un symbole, le mode :

(c1, ((c2, (mb1, 0)), ((c3, ((c4, (mb2, (mb3, 0))), 0)), (m1, 0))))

aura une représentation dont la forme peut être schématisée de la façon suivante :

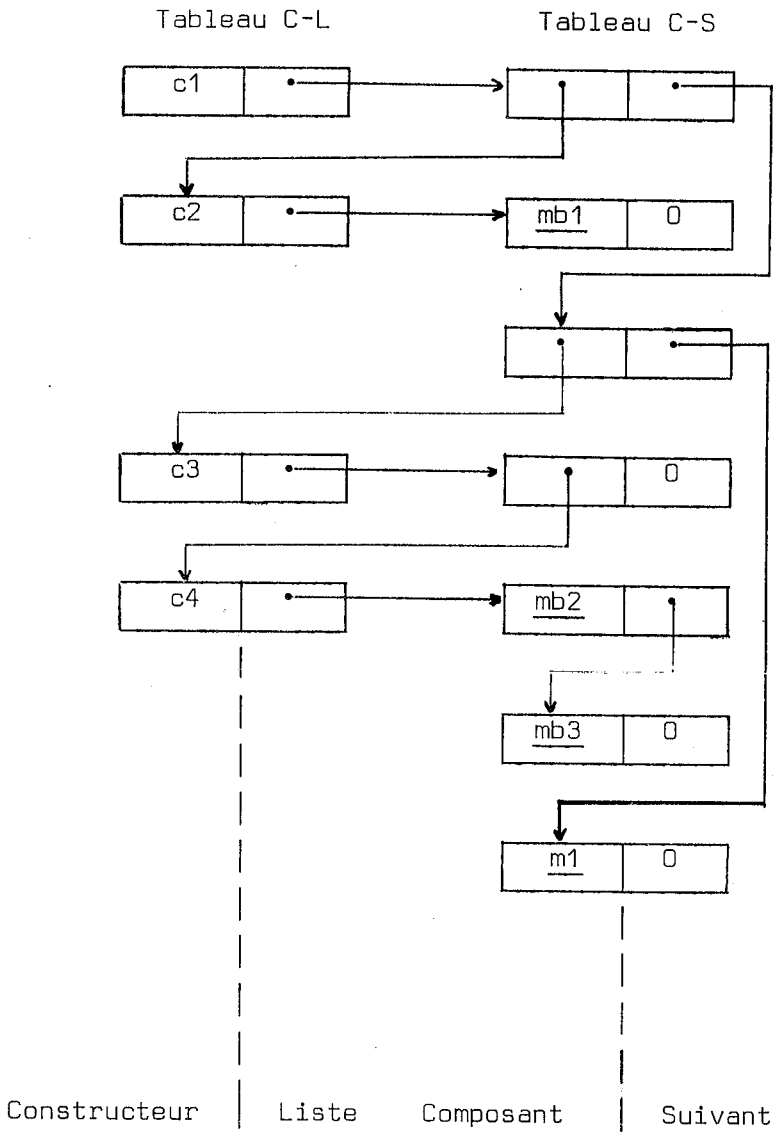


Avec c1 = tuple, c2 = ptr, c3 = seq, c4 = proc, mb1 = int, mb2 = real et mb3 = bool, cette liste est une représentation de :

tuple [ptr int, seq proc [real] bool, m1]

Il est à noter que dans le cas du constructeur struct, les sélecteurs qui identifient les champs font également partie du mode construit. Cependant, en ce qui concerne la représentation, il ne s'agit que d'un problème secondaire et cette particularité de struct sera omise dans tout ce qui suit afin de simplifier l'exposé.

En ce qui concerne l'implémentation, les listes représentant les modes de type ml sont construites à l'aide de deux tableaux appelés C-L, pour Constructeur-Liste, et C-S, pour Composant-Suivant. Ainsi, le mode de l'exemple précédent sera représenté en mémoire de la façon suivante :



Donc, chaque mode construit de type ml est représenté par une entrée dans le tableau C-L. Comme on le verra par la suite, c'est dans ces tableaux que l'on cherchera à éliminer les représentations multiples d'un même mode construit. Par exemple, pour le mode tuple [ptr int, seq ptr int], il n'y aura qu'une représentation du mode ptr int, bien qu'il soit utilisé deux fois.

I.2.2 - c = union

Les modes construits avec union sont de type mu. Avant de décrire la façon dont ils sont représentés, il est nécessaire d'introduire certaines propriétés de ces modes :

a) - Commutativité -

Si m1, m2, ..., mn sont des modes, on a l'identité suivante :
union [m1, ..., mi, mj, ..., mn] = union [m1, ..., mj, mi, ..., mn] ;
i, j ∈ [1, n]

b) - Simplification -

union [m1, ..., mi, mj, mk, mn] = union [m1, ..., mi, mk, ..., m
si mi = mj ; i, j ∈ [1, n]

c) - Remplacement -

union [m1, ..., mi, ... mn] = union [m1, ..., mi1, mi2, mip, ... mn]
si mi = union [mi1, ..., mip] ; i ∈ [1, n]

De plus, il faut noter que :

union [m] ≠ m si m ≠ union [m1, m2, ..., mn]

Représentation des modes de type mu

Un mode type mu peut être vu comme un vecteur dont les composantes sont d'autres modes, de type md, mb, ml ou mu. On peut d'ailleurs remarquer que si on a tenu compte de la propriété *c* (remplacement) aucune composante n'est de type mu, ni même de type md tel que md rep union [---].

D'autre part, il est possible de définir un ordre sur l'ensemble des modes qui ont été construits dans un programme. On peut donc représenter un mode de type mu par un vecteur booléen dans lequel la $j^{ème}$ composante, comp(j), est interprétée comme suit :

comp(j) = 1 indique que le $j^{ème}$ mode appartient à mu
comp(j) = 0 indique que le $j^{ème}$ mode n'appartient pas à mu

De plus, afin de simplifier le travail ultérieur sur les modes de type mu, l'ordre sur l'ensemble des modes peut être choisi de telle façon

que chaque vecteur représentant une union puisse être divisé en "tranches". C'est ainsi que l'on définira quatre intervalles d'entiers I1, I2, I3 et I4 tels que :

$$\begin{cases} I1 = [i_{11}, i_{1m}] \\ I2 = [i_{21}, i_{2n}] \text{ avec } i_{21} = i_{1m} + 1 \\ I3 = [i_{31}, i_{3p}] \text{ avec } i_{31} = i_{2n} + 1 \\ I4 = [i_{41}, i_{4q}] \text{ avec } i_{41} = i_{3p} + 1 \end{cases}$$

Etant donnée la $j^{ème}$ composante d'un vecteur représentant un mode de type mu, on associera alors à ces intervalles les interprétations suivantes :

$$\begin{cases} \text{si } j \in I1, \text{ comp(j) correspond à un mode de type md} \\ \text{si } j \in I2, \text{ comp(j) correspond à un mode de type mb} \\ \text{si } j \in I3, \text{ comp(j) correspond à un mode de type ml} \\ \text{si } j \in I4, \text{ comp(j) correspond à un mode de type mu} \end{cases}$$

Finalement, l'ensemble des vecteurs représentant des modes de type mu sera rangé dans un tableau appelé M-U, qui est donc une matrice booléenne dont les lignes sont les vecteurs considérés, et qui a $i_{4q} - i_{11} + 1$ colonnes.

Donc :

$$\begin{cases} \text{si } j \in I1, \text{ on accède au mode correspondant par la table M-D} \\ \text{si } j \in I2, \text{ le mode correspondant est l'un des modes de base} \\ \text{si } j \in I3, \text{ le mode correspondant est représenté dans C-L} \\ \text{si } j \in I4, \text{ le mode correspondant est représenté dans M-U} \end{cases}$$

D'autre part, il est clair que :

$$\begin{cases} i_{1m} - i_{11} + 1 = \text{nombre maximum d'entrées dans M-D} \\ i_{2n} - i_{21} + 1 = 5 \\ i_{3p} - i_{31} + 1 = \text{nombre maximum d'entrées dans C-L} \\ i_{4q} - i_{41} + 1 = \text{nombre maximum de vecteurs dans M-U} \end{cases}$$

On pourra voir par la suite que la représentation de l'ensemble des modes de type mu par une matrice booléenne apporte une simplification considérable aux programmes de traitement des modes.

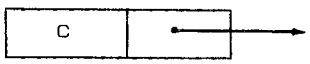
II - PROPRIETES ELEMENTAIRES DES REPRESENTATIONS -

Les définitions qui sont données ici introduisent la terminologie et les algorithmes élémentaires qui seront utilisés par la suite dans l'étude des problèmes de l'équivalence et de la réduction des représentations des modes.

II.1 - Cas des modes de type ml

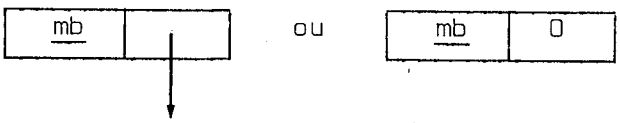
a) - Dans la représentation sous forme de liste binaire des modes de type ml, on distingue trois catégories de noeuds :

1) - Les noeuds de la forme (constructeur, liste). Ce sont ceux qui sont représentés dans le tableau C-L. Ils ont la forme :



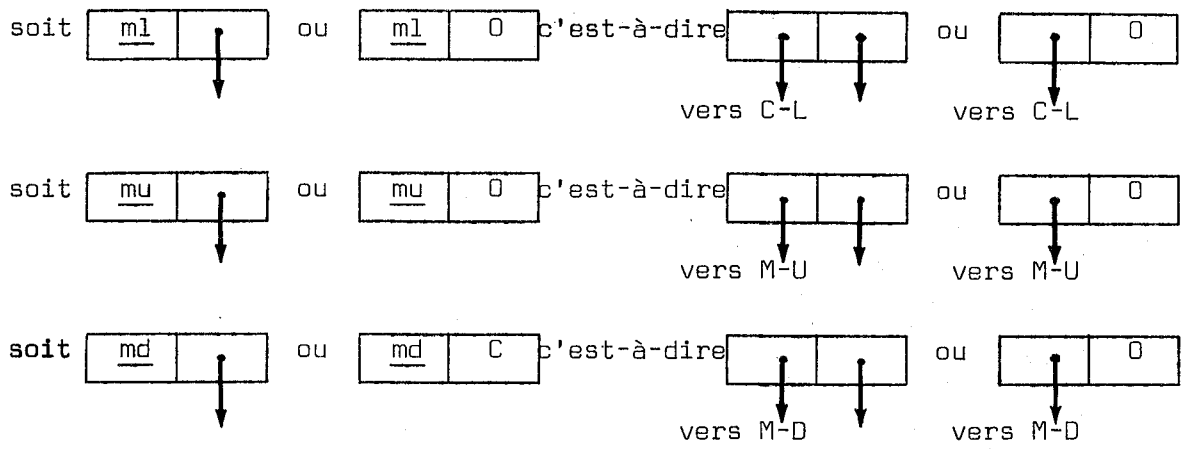
On les appellera des noeuds-c.

2) - Les noeuds représentés dans le tableau C-S, et dont le "composant" est un mode de base :



On les appellera des noeuds-b.

3) - Les noeuds, représentés dans le tableau C-S, et dont le "composant" est soit un mode de type ml, soit un mode de type mu, soit un mode de type md. La partie "composant" de ces noeuds fait donc référence soit à un élément de C-L, soit à un vecteur de M-U, soit à un élément de M-D :



Ces noeuds seront appelés des noeuds-r.

b) - Par la suite, lorsque l'on étudiera l'équivalence des modes, on sera amené à comparer deux représentations. L'un des éléments de cette comparaison est la comparaison de deux noeuds, dont l'égalité est définie de la façon suivante :

- deux noeuds-c sont égaux si leurs constructeurs sont identique
- deux noeuds-b sont égaux si leurs composants sont identiques.
- deux noeuds-r sont égaux si les représentations que référencent leurs composants sont équivalentes (l'équivalence de deux représentations sera définie ultérieurement dans Ch.2, III.1).

c) - De plus, la comparaison de deux modes de types ml va faire appel à la notion classique de parcours d'une liste : parcourir une liste consiste à en énumérer tous les noeuds dans un certain ordre. Dans la pratique, le parcours d'une liste est représenté par un processus récursif qui utilise une pile pour retenir les accès aux sous-listes dont le parcours n'a pas encore été fait : le parcours se termine lorsque la pile est vide. De plus, il faut noter que l'existence de modes récursifs va créer des boucles dans les représentations : ceci nécessite un processus de marquage des noeuds au cours d'un parcours afin de ne pas repasser indéfiniment par le même chemin.

d) - La relation d'égalité entre deux noeuds et la notion de parcours d'une liste permettent maintenant de définir l'identité entre deux listes l1 et l2.

Soient L1 l'ensemble des noeuds de l1 et L2 l'ensemble des noeuds de l2. Les parcours de l1 et l2 définissent une relation d'ordre sur L1 et sur L2. Si n_i et n_j sont deux noeuds de L1 ou deux noeuds de L2, on écrira :

$n_i < n_j$ si n_i a été rencontré avant n_j pendant le parcours de la liste à laquelle appartiennent n_i et n_j.

Soient L'1 et L'2 les ensembles L1 et L2 munis de leur relation d'ordre.

On dira alors qu'il y a identité entre les listes l1 et l2, et on écrira l1 $\equiv$ l2 si et seulement si il existe une bijection f de L'1 sur L'2 telle que :

(1) - si $n_i \in$ L'1 et $n_j \in$ L'1 avec $n_i < n_j$, alors :

$f(n_i) \in$ L'2 et $f(n_j) \in$ L'2 avec $f(n_i) < f(n_j)$

(2) - et pour tout $n_i \in$ L'1, n_i étant un noeud-c, alors :

$f(n_i) \in$ L'2, $f(n_i)$ est un noeud-c avec $f(n_i)=r$

(3) - et pour tout $n_i \in$ L'1, n_i étant un noeud-b, alors :

$f(n_i) \in$ L'2, $f(n_i)$ est un noeud-b avec

$f(n_i) = n_i$

e) - Finalement, toujours pour les modes de type ml, on dira qu'une partie d'une liste est toute sous-liste contenue dans cette liste.

Par exemple :

ptr int, seq proc real et proc real sont des parties de tuple [ptr int, seq proc real]

De plus, proc real est lui même une partie de seq proc real.

II.2 - Cas des modes de type mu

a) On a vu qu'un mode de type mu est représenté par un vecteur booléen. On note comp(j) la $j^{ème}$ composante d'un tel vecteur, et on distinguera deux type de composantes :

1) - Les composantes qui correspondent à un mode de base, c'est-à-dire telles que $j \in I2$. Une telle composante sera notée comp-b.

2) - Les composantes telles que $j \in I1 \cup I3 \cup I4$. Les composantes font références à des modes représentés par ailleurs, soit dans M-D, soit dans C-L, soit dans M-U. Une telle composante sera notée comp-r.

b) - On définit également ce qu'est le parcours d'un vecteur : c'est l'énumération une à une, de toutes ses composantes, dans un certain ordre et leur parcours dans le cas de comp-r. C'est-à-dire : une composante, si elle est égale à 1, peut entraîner le parcours d'une liste ou même d'un autre vecteur, s'il agit d'une composante de type comp-r. Le parcours d'un vecteur est donc un processus récursif qui nécessite l'utilisation d'une pile et un marquage du trajet déjà suivi pour arrêter le parcours dans le cas d'un mode récursif.

c) - L'identité de deux vecteurs V1 et V2, notée V1 ≡ V2, est définie de la façon suivante :

V1 est identique à V2 si et seulement si comp(j) de V1 ≡ comp(j) de V2, $j \in I1 \cup I2 \cup I3 \cup I4$.

d) - Finalement, on peut dire qu'une partie d'un vecteur est toute composante de ce vecteur.

II.3 - Cas général des modes construits -

a) - D'une façon générale, on appellera simplement parcours le parcours d'une liste ou le parcours d'un vecteur. On peut d'ailleurs remarquer que le parcours d'une liste peut entraîner le parcours d'un vecteur si cette liste contient un noeud-r faisant référence à M-U, et on a vu que le parcours d'un vecteur peut entraîner le parcours d'une liste.

b) - On appellera simplement partie d'un mode la partie d'une liste si ce mode est de type m1 ou la partie d'un vecteur si ce mode est de type mu. Une partie d'un mode quelconque m sera notée :

$$p(\underline{m})$$

et on a la relation :

$$p(\underline{m}) \subset \underline{m}$$

au niveau des représentations.

c) - L'identité entre deux modes m1 et m2 est soit l'identité de deux listes, si m1 et m2 sont de type m1, soit celle de deux vecteurs si m1 et m2 sont de type mu.

d) - Pour exprimer qu'un mode m dépend d'un mode de type md, on introduit une notation dite notation de fonction, qui est définie comme suit :

Soit m un mode. Si {md1, md2, ..., mdn} est l'ensemble des modes de type md qui sont référencés par des noeuds-r ou des comp-r dans la représentation de m, on exprimera que m dépend des symboles md1, ..., mdn, en écrivant :

$$\underline{m} = \varphi\ (\underline{md1}, \ldots, \underline{mdn})$$

Si $\underline{m} = c(\underline{m1}, \ldots, \underline{mn})$ et si $\underline{m} = \varphi\ (\underline{mi})$ avec $1 \leqslant i \leqslant n$, on dira que m fait référence directe à mi. De plus si, à son tour, $\underline{mi} = \varphi\ (\underline{mij})$,

on dira que m fait référence indirecte à mij. Par la suite, lorsqu'on dira qu'un mode fait référence à un autre mode de type md, cela voudra dire aussi bien référence directe que référence indirecte.

e) - Les diverses notations qui viennent d'être définies vont permettre maintenant d'exprimer simplement une des règles importantes de BASEL relative aux déclarations de modes. Cette règle sert à interdire la déclaration de modes auxquels ne correspondrait aucune valeur, comme :

m1 rep m2

m2 rep m1

Elle sert également à éviter de déclarer des modes dont les valeurs correspondantes occuperaient une place infinie en mémoire, comme :

m3 rep tuple [int,m3]

En effet, une valeur de mode m3 demande un emplacement pour un entier, plus un emplacement pour une valeur de mode m3, c'est-à-dire, encore un emplacement pour un entier, etc....

Par contre, si on déclare :

m4 rep tuple [int, ptr m4]

les valeurs de mode m4 occupent un emplacement de taille bien définie : un emplacement pour un entier, plus un emplacement pour un pointeur, c'est-à-dire pour une adresse.

Exprimée à l'aide des définitions précédentes, la règle est alors la suivante : si m est de type md et a été déclaré par :

m rep c (m1, ..., mn)

et si m = φ (m), tout parcours qui va de m à un noeud-r ou à un comp-r faisant référence à m doit contenir au moins un noeud-c ayant ptr comme constructeur.

D'après cette règle, il est donc interdit d'écrire :

m rep proc [m] none

ou

m rep tuple [union [int, seq m], m, bool]

Par contre, on peut écrire :

m1 rep struct [ptr m2 s1, ptr m2 s2]

m2 rep union [char, m1]

m3 rep struct [int val, ptr m3 proche]

On peut même écrire :

m4 rep ptr m4

Les valeurs de mode m4 peuvent être représentées (ce sont de simples adresses) mais leur utilité semble assez restreinte !

III - EQUIVALENCE DES MODES -

La relation d'équivalence entre deux modes est la relation fondamentale à partir de laquelle sont construites la plupart des algorithmes généraux de traitement des modes pendant la compilation.

III.1 - Définition de l'équivalence -

Deux modes m1 et m2 sont dits équivalents, et on notera m1 $\doteq$ m2, s'ils vérifient l'un des cas suivants :

(1) m1 est de type mb et m2 est de type mb ; alors m1 $\doteq$ m2 si et seulement si m1 = m2.

(2) ou, m1 est de type ml et m2 est de type ml,
avec m1 = c1(m11, ..., m1n) et m2 = c2(m21, ..., m2n) ;
alors m1 $\doteq$ m2 si et seulement si :

$$c1 = c2$$

et

$$m1i = m2i \text{ pour } i = 1, 2, \ldots n.$$

(3) ou, m1 est de type mu et m2 est de type mu,
avec m1 = union [m11, ..., m1p] et m2 = union [m21, ..., m2q]
tels que m1i ne soit pas de type mu pour i = 1, 2, ...p et
m2j ne soit pas de type mu pour j = 1, 2, ...q, alors m1 $\doteq$ m2
si et seulement si :

pour tout $i \in [1, p]$, il existe $j \in [1, q]$ avec m1i $\doteq$ m2j

et pour tout $j \in [1, q]$, il existe $i \in [1, p]$ avec m2j $\doteq$ m1i

Remarque : pour les modes md, l'équivalence est définie d'après les modes qu'ils représentent (mb, ml ou mu).

Dans tous les autres cas : m1 et m2 ne sont pas équivalents,
et on notera m1 $\not\doteq$ m2.

III.2 - Intérêt de la relation d'équivalence -

C'est la connaissance de l'équivalence entre deux modes qui va permettre d'éviter la duplication de représentations redondantes et d'étudier les possibilités de réduction des représentations. En effet :

1) - Si m1 $\doteq$ m2 on a intérêt à ne pas conserver une représentation pour m1 et une autre pour m2, mais une seule pour tous les deux.

2) - Si $p_1(m)$, $p_2(m)$, ..., $p_n(m)$ sont des parties d'un même mode m, telles que : m $\doteq$ $p_1(m)$ $\doteq$ ... $\doteq$ $p_n(m)$. (C'est possible si m est défini de façon récursive), on a intérêt à rechercher la partie $p_i(m)$ dont la représentation est la plus "synthé-

tique", et à conserver cette seule représentation comme représentation de m.

De plus, afin de simplifier le travail sur les modes au moment du traitement de la partie instructions, et de rendre aussi simple que possible le test attaché à une déclaration conditionnelle, on devra rechercher une représentation canonique pour les modes de type mu à partir de laquelle il ne sera plus nécessaire de faire appel aux propriétés de commutativité, de simplification et de remplacement des unions.

IV - ALGORITHMES DE RECHERCHE DE L'EQUIVALENCE -

On présente d'abord ici un algorithme général pour la recherche de l'équivalence entre deux modes. Puis, à la lumière d'une analyse détaillée du fonctionnement de cet algorithme, on construira un algorithme simplifié qui donne les mêmes résultats que l'algorithme général avec de bien meilleures performances.

IV.1 - L'Algorithme Général (A. G.) -

L'algorithme présenté ici utilise les représentations définies plus haut. Son but est le suivant : étant donnés deux modes m1 et m2 dont on a les représentations, rechercher si ces représentations correspondent à deux modes équivalents, c'est-à-dire si m1 $\doteq$ m2.

IV.1.1 - Présentation de l'Algorithme Général -

On distingue trois cas généraux dans le fonctionnement de l'algorithme :

1er Cas. Si m1 est de type mb et si m2 est de type mb, il suffit de voir si m1 $\equiv$ m2.

2ème Cas. Si m1 est de type ml et si m2 est de type ml, on effectue un parcours de la liste représentant m1 et, simultanément, on effectue aussi un parcours de la liste représentant m2, en comparant les noeuds-b et les noeuds-c rencontrés de part et

d'autre à chaque pas de ce double parcours. Si les deux parcours se terminant simultanément, en ayant constaté l'égalité des noeuds-b et des noeuds-c rencontrés de part et d'autre, alors m1 ≐ m2.

Par contre, si on a trouvé une inégalité entre deux noeuds-b ou entre deux noeuds-c, l'un appartenant à m1 et l'autre à m2, ou si un parcours se termine alors que l'autre n'est pas terminé, alors m1 ≠ m2.

Ce 2ème cas peut se formaliser de la façon suivante :

- soit l1 la liste qui représente m1 ;
- soit l2 la liste qui représente m2 ;
- soit L'1 l'ensemble, ordonné par le parcours, des noeuds de l1
- soit L'2 l'ensemble, ordonné par le parcours, des noeuds de l2.

Alors : m1 ≐ m2 si et seulement si il existe une application f de L'1 sur L'2 telle que, si n_i, n_j et n_k sont des noeuds quelconques appartenant à L'1 :

- si $n_i < n_j$, alors $f(n_i) < f(n_j)$,
- et si n_k est de type noeud-c, alors $f(n_k) = n_k$,
- et si n_k est de type noeud-b, alors $f(n_k) = n_k$.

De plus, on peut noter que les propriétés de l'application f ne peuvent pas toutes être connues de façon générales. En effet, si l'on sait qu'elle est toujours surjective, elle ne sera biunivoque que dans le cas où les listes sont identiques. Or, deux modes peuvent être équivalents sans que les listes qui les représentent soient identiques : ceci se rencontre pour des modes m1 et m2 définis de façon récursive ; par exemple :

m1 rep ptr m1

m2 rep ptr ptr m2

3ème Cas. Si m1 est de type mu et si m2 est de type mu, on effectue d'abord un parcours de m1, et pour toute composante comp1(i) de m1 avec comp1(i) = 1, on recherche dans m2 une composante comp2(j) = 1 telle que le mode correspondant à comp2(j) soit équivalent au mode correspondant à comp1(i). Si une telle composante comp2(j) n'existe pas, m1 $\not\doteq$ m2. Par contre, si pour toutes les composantes comp1(i) on a pu trouver une composante comp2(j) qui satisfasse les conditions, alors on intervertit les rôles de m1 et m2 dans le processus, et on recommence la comparaison. Si après ces deux passages on a toujours trouvé une composante satisfaisante dans le deuxième mode, alors m1 $\doteq$ m2.

Ce troisième cas peut être formalisé de la façon suivante :

- soit M(comp(i)) le mode correspondant à une composante,
- soit L1 l'ensemble des comp1(i) de m1,
- soit L2 l'ensemble des comp2(j) de m2.

On suppose que L1 et L2 sont tels qu'aucune composante ne correspond à un mode de type mu, c'est-à-dire que l'on a tenu compte de la propriété de remplacement dans les unions : cela est en effet toujours possible (Voir III.2.2.e).

Soit maintenant :

- L'1 l'ensemble L1 muni de la relation d'ordre définie par le parcours de m1.
- L'2 l'ensemble L2 muni de la relation d'ordre définie par le parcours de m2.

Alors, m1 est équivalent à m si et seulement si :

1) - Il existe une application f de L'1 dans L'2 telle que :

• pour tout comp1(i) $\in$ L'1 avec i $\in$ I2,
M(f(comp1(i))) $\equiv$ M(comp1(i))

• et pour tout comp1(i) $\in$ L'1 avec i $\in$ I1 U I3,
M(f(comp1(i))) $\doteq$ M(comp1(i)) ;

2) - et il existe une application g de L'2 dans L1, telle que :

- pour tout comp2(j) $\in$ L'2 avec j $\in$ I2,
  M(g(comp2(j))) $\equiv$ M(comp2(j))
- pour tout comp2(j) $\in$ L'2 avec j $\in$ I1 U I3,
  M(g(comp2(j))) $\doteq$ M(comp2(j)).

Cette formalisation tient compte du fait que plusieurs représentations pour des modes équivalents ont pu être conservées, par exemple dans C-L. Par contre, si l'on a tenu compte des diverses propriétés des unions (commutativité, simplification, remplacement) et si tous les modes de type ml sont rerpésentés de façon unique, c'est-à-dire que l'on n'a pas dans C-L deux modes équivalents, la recherche de l'équivalence entre deux modes de type mu se réduit à la comparaison de deux vecteurs booléens : c'est là l'intérêt de la représentation choisie pour les modes de type mu, et l'un des avantages essentiels de la recherche d'une représentation unique pour chaque mode.

Finalement, il faut noter que, dans le cas général, le processus du deuxième cas peut faire appel à celui du troisième cas, et vice-versa. De plus, chacun d'eux fait ultimement appel au cas 1 qui traite les modes de base, c'est-à-dire les éléments "terminaux" des représentations.

IV.1.2 - Analyse de l'Algorithme Général

On va examiner ici le fonctionnement de l'algorithme qui vient d'être présenté, dans le cas où les modes à comparer sont définis de façon récursive, c'est-à-dire tels que $\underline{m} = \varphi(\underline{m})$. A ce sujet, voir aussi [KO]. En effet, c'est dans ce cas que des simplifications vont apparaître, qui diminueront sensiblement le nombre de pas élémentaires à effectuer lors d'une recherche d'équivalence. Pour cela, quelques remarques sur le parcours des modes définis de façon récursive sont d'abord nécessaires.

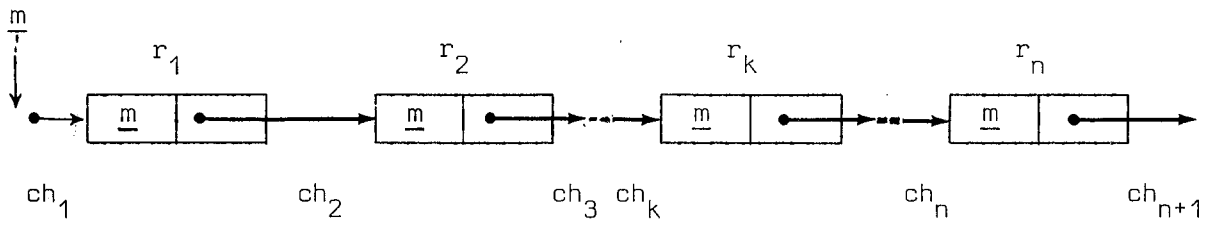
IV.1.2.1 - Parcours des modes $\underline{m}$ tels que $\underline{m} = \varphi(\underline{m})$.

Chaque fois que l'on trouve une référence pendant le parcours de la représentation d'un mode, c'est-à-dire chaque fois que l'on rencontre un noeud de type noeud-r dans un mode de type ml ou une composante de type

comp-r dans un mode de type mu, il faut effectuer le parcours de la représentation ainsi référencée, en laissant dans une pile une indication de l'endroit où le parcours précédent a été interrompu, afin de pouvoir le reprendre lorsque le parcours de la représentation référencée sera terminé.

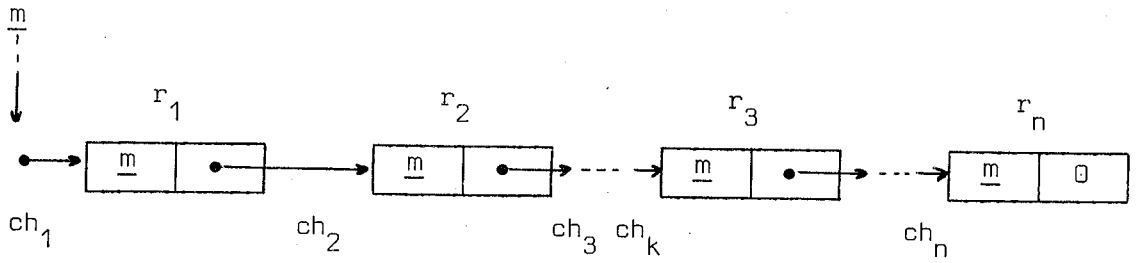
Donc, lorsque l'on doit parcourir la représentation d'un mode m tel que m = φ (m), chaque fois que l'on trouve une référence à m, le parcours de la représentation ainsi référencée implique un retour au point de départ du parcours d'origine. La question qui se pose est alors la suivante : parcourt-on plusieurs fois la représentation de m et, si oui, combien de fois ?

Pour étudier ce problème, la représentation de m sera schématisée de la façon suivante où ne sont distingués que les noeuds-r qui font référence à m :



Quelques remarques sur cette schématisation :

- Il est évident que les cheminements ch_1, ch_2, ..., ch_{n+1}, qui sont représentés ici par de simples flèches, peuvent être fort complexes, et que leur parcours peut entraîner un certain nombre d'opérations sur la pile. Cependant, on ne détaillera pas ici le contenu de ces cheminements, pour s'intéresser aux noeuds-r r_1, r_2, ..., r_n qui sont des références au mode de départ.
- Le cheminement ch_{n+1} conduit de la dernière référence à m à la fin du parcours de m. Comme tous les autres cheminements, ch_{n+1} ne contient pas de référence à m. C'est pourquoi, en n'enlevant rien à la généralité du problème, on peut supprimer ce cheminement dans le schéma de la représentation à parcourir :



Mais on ne peut évidemment supprimer aucun des autres cheminements car ce sont eux qui conduisent aux références à m qui sont à l'origine du problème posé.

On peut maintenant représenter le processus du parcours de la représentation de m par une procédure appelée PARCOURS. Cette procédure utilise 3 variables globales et une pile. Les variables utilisées sont :

i : la position, dans la représentation, d'un cheminement ch_i, i est aussi utilisé pour indiquer, dans la pile, où on a dû interrompre le parcours pour repartir au début de la représentation à cause d'une référence à m. C'est initialisé à 1 au début du processus.

j : indice permettant d'effectuer les parcours impliqués par les références à m.

n : nombre de références à m, qui est supposé connu d'avance, car on veut seulement examiner ici comment s'effectue le parcours pour une valeur donnée de n.

D'une façon informelle, on peut alors décrire la procédure PARCOURS comme suit :

PARCOURS

A - Parcourir et marquer ch_i ;

B - Arrivée à r_i ; marquer r_i ;

C - Mettre i au sommet de la pile ;

D - $j \leftarrow 1$; revenir au départ, c'est-à-dire au début de la représentation de $\underline{m}$;

D-1 - Parcourir ch_j, qui est déjà marqué ;

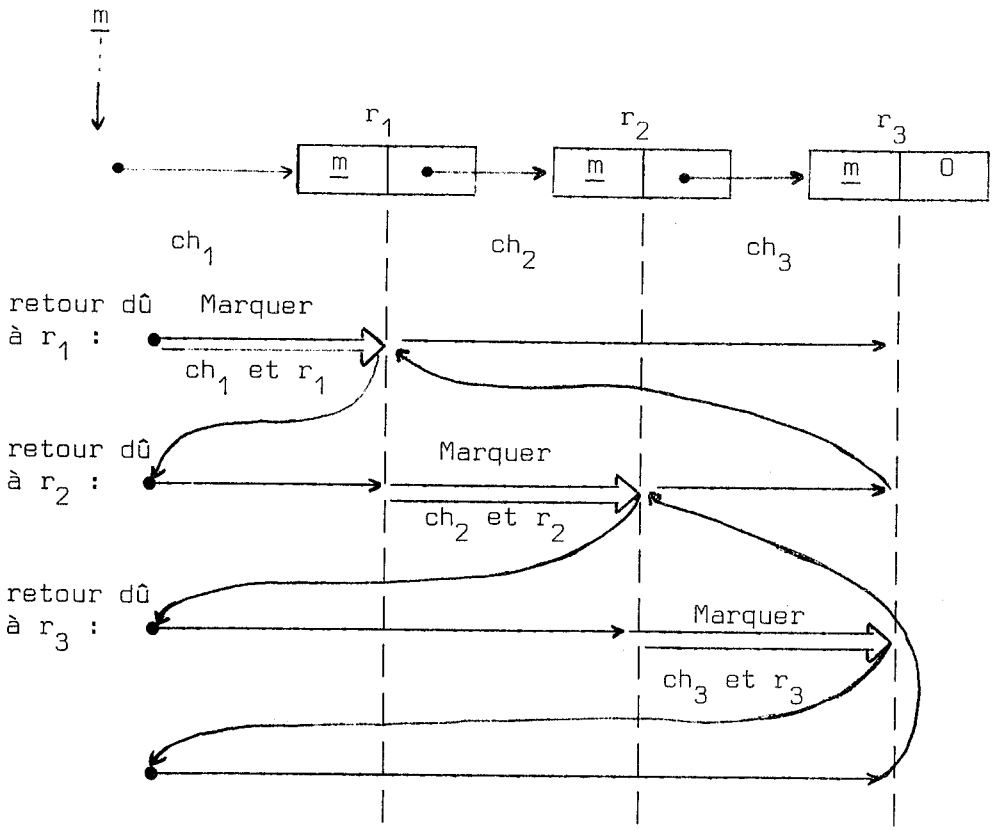
D-2 - Arrivée à r_j, qui est déjà marqué, ce qui signifie qu'on ne doit pas revenir au départ ;

D-2-1 - Si $j < i$, continuer le parcours, c'est-à-dire : $j \leftarrow j + 1$; aller à D-1 ;

D-2-2 - Si $j = i$ et $i \neq n$:
- $i \leftarrow i + 1$;
- appeler PARCOURS ;
- supprimer le sommet de la pile ; soit l la valeur qu'il contenait ;
- parcourir ch_{l+1}, ..., ch_n, qui sont déjà marqués ;
- sortir de PARCOURS ;

D-2-3 - Si $j = i$ et $i = n$:
- supprimer le sommet de la pile (c'est-à-dire n) ;
- sortir de PARCOURS.

Par exemple avec n = 3 on peut voir que le parcours s'effectuera de la façon indiquée en dessous du schéma de la représentation :



On peut maintenant répondre à la question suivante : pour une valeur de k entre 1 et n, combien de fois parcourt-on ch_k ?

1) - D'après A -, on parcourt ch_k une première fois, en le marquant ;

2) - D'après D-2-1, pour tout j tel que $k \leqslant j \leqslant n$, on parcourt ch_k une fois en le trouvant déjà marqué. Donc, sur l'ensemble du parcours, D-2-1 implique n-k+1 parcours de ch_k.

3) - D'après D-2-2, lorsqu'on repart du cheminement indiqué par le sommet de la pile, on parcourt ch_k une fois pour chaque j tel que $1 \leqslant j \leqslant k-1$. Donc, en tout, D-2-2 implique k-1 parcours de ch_k.

Finalement, on voit que le nombre de passages sur ch_k est :

$$1 + (n-k+1) + (K-1) = n+1$$

Ainsi, il apparaît que s'il y a n références à m dans la représentation de m, un parcours de m effectuera n+1 passages sur cette représentation.

IV.1.2.2 - Comparaison de deux modes qui font référence à eux-mêmes -

On considère ici le cas où :

m1 = φ (m1), et m1 fait n fois référence à lui-même.

m2 = φ (m2), et m2 fait n fois référence à lui-même.

Ce serait le cas, par exemple, avec :

m1 rep tuple [ptr m1, ptr seq m1]

m2 rep tuple [ptr m2, ptr seq m2]

D'après ce que l'on vient de dire à propos du parcours de ce genre de mode, si m1 $\doteq$ m2, l'algorithme général de comparaison parcourra entièrement chacune de deux représentations, c'est-à-dire que l'on passera n+1 fois sur la représentation de m1 et n+1 fois sur celle de m2. Il est donc clair que l'utilisation de l'A. G. peut parfois être fort lourde, et une simplification de son fonctionnement est souhaitable pour éviter des parcours multiples d'une même représentation.

IV.2 - L'Algorithme Simplifié (A. S.)

Pour comparer m1 avec m2, le principe général reste le même que pour l'A. G. : un parcours des deux représentations conduit soit à la découverte d'une différence, cas où il n'y a pas équivalence, soit à la fin simultanées des deux parcours sans découverte de différence, cas où il y a équivalence.

IV.2.1 - Particularité de l'Algorithme Simplifié -

Les simplifications apportées à l'A. G. pour obtenir l'A. S. résident dans les deux remarques suivantes :

1) - Si on arrive à une référence à m1 (resp. m2) dans l'une des représentations, et, en même temps, à une référence à m1 (resp. m2) dans l'autre représentation, il est évident qu'il n'est pas nécessaire de revenir au point de départ ainsi indiqué, car on a trouvé la même information dans les deux représentations.

2) - Si on arrive à une référence à m1 (resp. m2) dans l'une des représentations, et, en même temps, à une référence à m2 (resp. m1) dans l'autre représentation, on suppose que m1 $\doteq$ m2. Ceci signifie que toute référence à m1 est considérée comme représentant la même information que toute référence à m2 : tout se passe alors comme dans le cas précédent, et on ne revient pas aux points de départ des représentations de m1 et m2.

Dans ces conditions, on dira que m1 $\doteq$ m2 si on arrive simultanément à la fin de leurs représentations respectives sans avoir rencontré d'autres différences que celles mentionnées dans la deuxième remarque ci-dessus. Par contre, si une autre différence a été rencontrée (dans les ch_j), on a alors m1 $\not\doteq$ m2.

Ainsi, là où l'A. G. faisait n+1 parcours d'une représentation avec n références au mode de départ, l'A. S. ne fera qu'un seul parcours. Cependant, il est évident que l'A. S. parcourra les ch_j exactement de la même façon que l'A. G., bien qu'il ne le fasse qu'une fois. La seule différence entre l'A. G. et l'A. S. réside dans un traitement différent des noeuds r_1, r_2, ..., r_n.

IV.2.2 - Validité de l'Algorithme Simplifié -

Pour montrer que l'A. S. établit bien l'équivalence de deux modes. Il suffit de montrer :

(1) - si m1 $\not\doteq$ m2, l'A. S. trouve m1 $\not\doteq$ m2.

(2) - si l'A. S. trouve m1 $\not\doteq$ m2, alors m1 $\not\doteq$ m2 [KN]

Dans la démonstration qui suit, on suppose que m1 est comparable avec m2, c'est-à-dire soit que si m1 est de type ml, alors m2 est de type ml, soit que si m1 est de type mu, alors m2 est de type mu.

(1) - Si m1 $\not\doteq$ m2, il existe au moins une différence entre les deux représentations. Cette différence peut être :

- soit un noeud-b de m1 $\neq$ noeud-b correspondant de m2 ;
- soit un noeud-c de m1 $\neq$ noeud-c correspondant de m2 ;
- soit un ch_j de m1 $\neq$ ch_j correspondant de m2, pour un certain j ;
- soit une composante $\in$ I2 dans m1 $\not\in$ toute composante $\in$ I2 dans m2 (ou vice-versa) ;
- soit une composante I1 U I3 dans m1 $\not\doteq$ toute composante I1 U 12 dans m2 (ou vice-versa). D'après l'A. G., ce cas se ramène d'ailleurs à l'un des quatre cas précédents.

Si une telle différence existe, l'A. S., qui assure un parcours total de chaque représentation, va la trouver, et le fait de supposer m1 $\doteq$ m2 quand on rencontre une référence à m1 et une référence à m2 ne peut en aucun cas empêcher la découverte de cette différence. En effet, le raisonnement que l'on fait dans ce cas est le suivant :

Si l'on est parvenu jusqu'à une référence à m1 dans une représentation et à m2 dans l'autre, c'est qu'aucune différence n'a été découverte auparavant. Il n'est donc pas nécessaire de reparcourir les chemins que l'on vient de suivre, car on est sûr qu'aucune différence ne s'y trouve. On peut donc poursuivre les parcours car la différence recherchée ne peut être qu'entre les références rencontrées et la fin des parcours. Dès que cette différence sera rencontrée, l'A. S. concluera m1 $\not\doteq$ m2.

(2) - Si l'A. S. trouve m1 $\not\doteq$ m2, ceci signifie qu'il existe une différence autre que les références à m1 et m2. L'A. G., que l'on peut considérer comme définissant la relation d'équivalence, trouvera nécessairement cette différence, car il fait un parcours total des représentations. L'A. G. concluera donc, comme l'A. S., que m1 $\not\doteq$ m2.

IV.3 - Exemples et remarques sur les algorithmes précédents -

D'après ce que l'on vient de voir, il est clair que l'on a toujours intérêt à utiliser l'A. S. plutôt que l'A. G.. Dans les exemples suivants, on peut noter les gains obtenus sur la longueur des parcours effectués.

1 - m1 rep ptr m1
m2 rep ptr m2
(m1 ≐ m2)

2 - m1 rep ptr ptr m2
m2 rep ptr m1
(m1 ≐ m2)

3 - m1 rep tuple [ptr m2, ptr m1]
m2 rep tuple [ptr m1, ptr m2]
(m1 ≐ m2)

4 - m1 rep tuple [ptr union [tuple [ptr union [m1, char], ptr union [m1, char]], char], ptr union [m1, char]]
liste rep tuple [ptr listelem, ptr listelem]
listelem rep union [liste, char]
(m1 ≐ liste)

5 - m1 rep tuple [ptr m3, ptr m2, ptr m4]
m2 rep tuple [ptr m4, ptr m1, ptr m3]
m3 rep ptr m4,
m4 rep ptr m3
(m1 ≐ m2 et m3 ≐ m4)

6 - m1 rep ptr ptr struct [m1 s] [CU]
m2 rep ptr m3
m3 rep ptr struct [m3 s]
(m1 ≠ m3)

7 - m1 rep ptr tuple [m3, m4, m2]
m2 rep ptr tuple [m4, m3, m1]
m3 rep ptr tuple [m2, m4]
m4 rep ptr tuple [m3, m1]
(m2 $\not\doteq$ m3, m3 $\not\doteq$ m4 et m3 $\not\doteq$ m1)

A titre d'exemple, les gains sur le parcours de chaque mode (nombre de parcours avec A. G. - nombre de parcours avec A. S.) sont égaux à (2-1), (2-1), (4-2) et (3-1) respectivement pour les exemples 1, 2, 3 et 5.

De plus, les trois derniers exemples (5, 6 et 7) vont permettre de faire des remarques intéressantes sur le problème du "marquage" pendant le parcours d'une représentation. En effet, il est important de bien choisir la nature des "marques" qui sont laissées par un premier passage sur une partie d'une représentation. Par exemple, les cas 6 et 7 montrent qu'une simple marque (un bit) n'est pas suffisante et peut même conduire à des conclusions erronées sur l'équivalence de deux modes. Dans le cas de l'exemple 6, si l'on se contente d'une simple marque pour comparer m1 et m2, on aura la séquence suivante des comparaisons :

- comparer m1 avec m2, c'est-à-dire :
→ comparer ptr avec ptr, et marquer ;
- comparer ptr struct [m1 s] avec m3, c'est-à-dire :
→ comparer ptr avec ptr, et marquer ;
- comparer struct [m1 s] avec struct [m3 s], c'est-à-dire
→ comparer struct avec struct, et marquer ;
- comparer m1 avec m3, c'est-à-dire :
→ comparer ptr avec ptr, mais ces deux ptr sont déjà marqués, ce qui signifie qu'il est inutile d'aller plus loin dans les listes qu'ils référencent. On serait donc conduit à conclure que m1 $\doteq$ m2, ce qui est faux. La raison en est que les marques sur le premier ptr de m1 et le premier ptr de m3 ne contiennent pas assez d'information : deux marques ne doivent arrêter le parcours que si elles ont été mises au même moment, ce qui n'est pas le cas des marques considérées ici.

Un mécanisme qui garde une trace des couples d'éléments traités simultanément est donc nécessaire pour interpréter correctement les marques faites sur ces éléments. De plus, l'utilisation d'une telle trace permet de découvrir des équivalences de modes autres que celle des modes pour lesquels l'algorithme de comparaison a été appelé. C'est le cas dans l'exemple 5 : en recherchant l'équivalence de m1 et m2, on découvre, au passage, l'équivalence de m3 avec m4. Grâce à la trace, ce résultat sera conservé au même titre que celui pour lequel la comparaison avait été faite à l'origine.

V - REPRESENTATIONS REDONDANTES ET REDUCTION -

Du fait qu'il est possible d'avoir des déclarations récursives pour les modes, on a vu que certains modes sont tels que $\underline{m} = \varphi(\underline{m})$. De plus, on peut toujours remplacer un symbole par le mode qu'il représente. Donc, si l'on a $\underline{m} = \varphi(\underline{m})$, c'est-à-dire :

$$\underline{m} = c(\ldots, \underline{m}, \ldots)$$

on a également :

$$\underline{m} = c(\ldots, c(\ldots, \underline{m}, \ldots), \ldots)$$

Or, cette deuxième façon d'écrire la définition de m conduit à une représentation plus complexe que la première, bien que ces deux représentations soient équivalentes. Le problème de la réduction consiste à simplifier de telles représentations redondantes dans le but d'obtenir la représentation la plus synthétique, que l'on appelle représentation minimale.

V.1 - Définitions relatives au problème de la réduction -

1 - MODE REDUCTIBLE : Un mode m est réductible (ou redondant s'il existe une partie $p(\underline{m}) \subset \underline{m}$ telle que $p(\underline{m}) \triangleq \underline{m}$.

MODE IRREDUCTIBLE : Un mode m est irréductible s'il n'existe pas de partie $p(\underline{m}) \subset \underline{m}$ telle que $p(\underline{m}) \triangleq \underline{m}$.

2 - PARTIE REDUCTIBLE : Une partie p(m) d'un mode m est réductible (ou redondante) s'il existe p(p(m)) $\subset$ p(m) telle que p(p(m)) $\doteq$ p(m).

PARTIE IRREDUCTIBLE : Une partie p(m) d'un mode m est irréductible s'il n'existe pas p(p(m)) $\subset$ p(m) telle que p(p(m)) $\doteq$ p(m).

3 - REPRESENTATION MINIMALE : Une partie p(m) d'un mode m (p(m) $\subset$ m) telle que p(m) $\doteq$ m et p(m) est irréductible est appelée représentation minimale de m. C'est la représentation la plus synthétique de m.

4 - REDUCTION D'UN MODE : Processus pour trouver la représentation minimale d'un mode. On dit que la représentation d'un mode est réduite quand elle est minimale.

V.2 - Exemples de modes redondants et de parties minimales -

1 - Soit la déclaration de mode :

m rep ptr tuple [ptr m, ptr tuple [ptr m, m]]

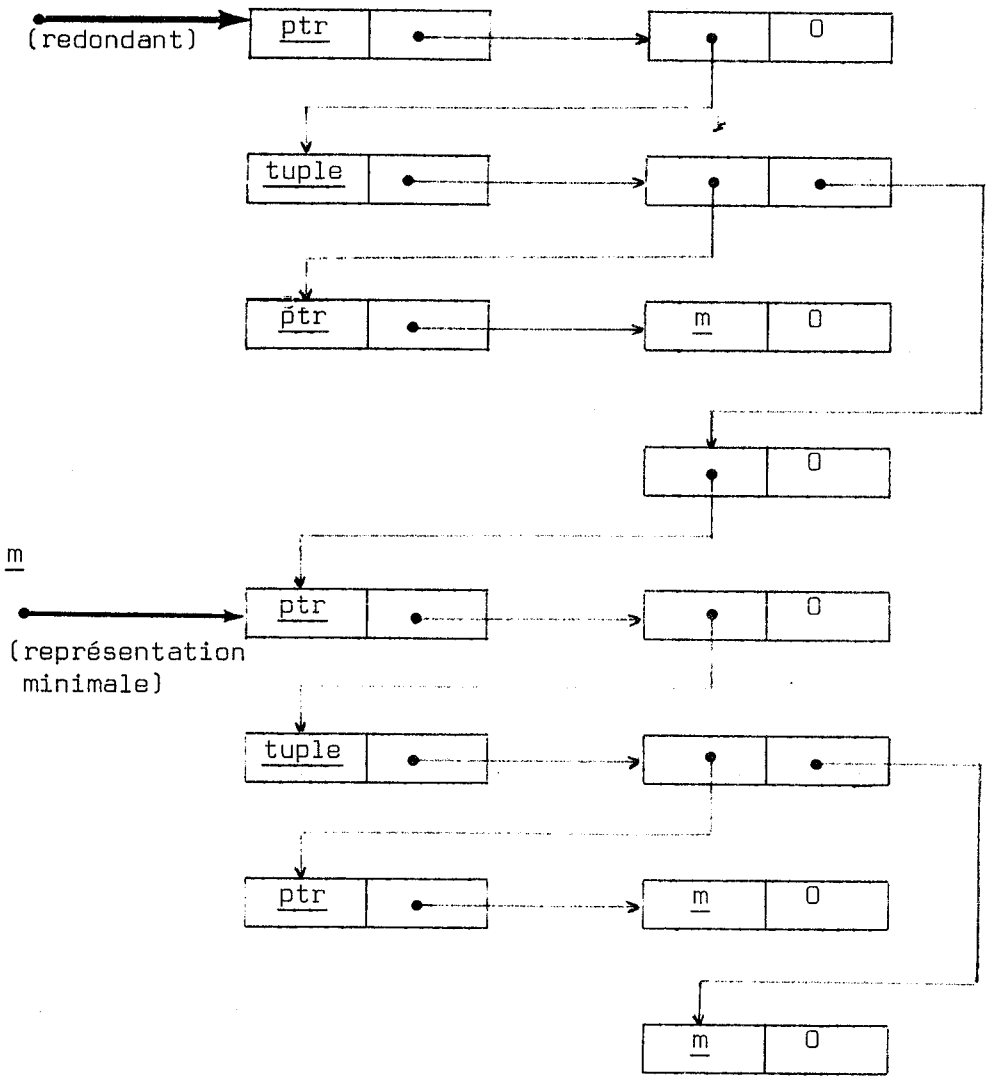
Une des parties de m est :

p(m) = ptr tuple [ptr m, m]

Et on a la relation : p(m) $\doteq$ m : m est donc redondant. De plus, on peut vérifier que p(m) ne contient pas de partie p(p(m)) telle que p(p(m)) $\doteq$ p(m).

Donc, p(m) est une représentation minimale de m. On a avantage à ne garder, pour représenter m, que la représentation de p(m). La simplification ainsi apportée est évidente quand on examine le schéma de ces représentations.

m



2 - Soient les déclarations :

m1 rep ptr union [ptr union [ptr m1, ptr m2, ptr m3], ptr union [ptr m1]],
m2 rep ptr union [ptr union [ptr m1, ptr m2], ptr union [ptr m2]],
m3 rep ptr union [ptr union [ptr m1]]

Dans ce cas, on a : m1 ≐ m2 ≐ m3 et m3 est la représentation minimale pour chacun de ces trois modes ; m3 devient :

m3 rep ptr union [ptr union [ptr m3]]

V.3 - L'Algorithme de Réduction -

Le but de l'algorithme de réduction est de trouver la représentation minimale d'un mode m à partir d'une représentation de ce mode.

Le principe de cet algorithme est le suivant :

- soit m le mode de départ.
- parmi toutes les parties p(m), en rechercher une équivalente à m.
- si on en trouve une, recommencer le processus en remplaçant m par cette partie.
- si on n'en trouve pas, c'est que l'on a une partie irréductible, donc minimale, ou que m est irréductible.

De façon plus rigoureuse, cet algorithme peut être décrit par une procédure REDUCTION (m) où m est le mode départ. Dans cette procédure, i est une variable utilisée pour faire une itération sur l'ensemble des parties. De plus, il faut rappeler que l'ensemble des parties d'un mode m, qui est construit à partir d'une représentation de ce mode, n'inclue pas cette représentation de m.

REDUCTION (m)

A - Soit $\mathcal{P}$ l'ensemble des parties de $\underline{m}$;

A-1 - Si $\mathcal{P}$: Ø, sortir, car $\underline{m}$ est alors irréductible, donc minimal,

B - Soit $\mathcal{P} = \{p_1(\underline{m}), p_2(\underline{m}), \ldots, p_n(\underline{m})\}$;
$i \leftarrow 1$;

C - Si $\underline{m} \doteq p_i(\underline{m})$;

C-1 - Remplacer $\underline{m}$ par $p_i(\underline{m})$, car $\underline{m}$ était redondant. On a donc fait un pas dans la réduction en simplifiant la représentation de $\underline{m}$;

C-2 - Appeler REDUCTION ($\underline{m}$) ;

C-3 - Sortir ;

D - Si $\underline{m} \neq p_i(\underline{m})$:

D-1 - Si $i \neq n$: $i \leftarrow i + 1$; aller à C ;

D-2 - Si $i = n$: sortir, car $\underline{m}$ est alors irréductible, donc minimal.

Etant donnée la façon dont fonctionne cet algorithme, on peut remarquer que la réduction se fait toujours en prenant la **pre**mière partie $p_i(\underline{m})$ qui a été trouvée équivalente au mode $\underline{m}$ de départ, et ceci à chaque étape du processus jusqu'à arriver à une partie irréductible, que l'on dit alors être minimale. Mais la question qui se pose est alors la suivante : peut-on être sûr que la partie irréductible ainsi trouvée est bien **la** représentation minimale du mode de départ ? Ou bien serait-il possible de trouver une autre partie irréductible, plus synthétique que celle fournie par cet algorithme, en prenant comme étapes intermédiaires de la réduction d'autres parties $p_i(\underline{m})$ que celles choisies ici ? Le problème est donc celui de l'unicité de la représentation minimale.

V.4 - Unicité de la représentation minimale -

Pour pouvoir montrer que la représentation minimale d'un mode est unique, il faut d'abord distinguer quelques propriétés de la relation d'équivalence entre deux modes.

V.4.1 - Propriétés de la relation d'équivalence -

Cette relation est :

1 - REFLEXIVE = $\underline{m} \doteq \underline{m}$.

En effet, il suffit de construire une fonction f : L' → L' telle que f(n) = n pour tout n ∈ L'.

2 - SYMETRIQUE : $\underline{m1} \doteq \underline{m2} \Rightarrow \underline{m2} = \underline{m1}$.

Si m1 = m2, il existe f : L'1 → L'2 telle que f est surjective, f conserve l'ordre et pour tout noeud-c (resp. noeud-b, comp-b) $n_1 \in$ L'1, $f(n_1) = n_1$ (voir Ch. 2, IV.1.1).

1er Cas. f est injective.

On peut alors construire g : L'2 → L'1 telle que $g(n_2) = f^{-1}(n_2)$ pour tout $n_2 \in$ L'2.

- g est surjective :
  Si $n_1 \in$ L'1, alors $f(n_1) = n_2 \in$ L'2, et $g(n_2) = n_1$.
  Tout élément de L'1 est donc le correspondant d'un élément de L'2 : g est surjective.

- g conserve l'ordre :
  Si n_{21}, $n_{22} \in$ L'2 et $n_{21} < n_{22}$, alors $g(n_{21}) < g(n_{22})$. En effet, si on suppose que $g(n_{21}) > g(n_{22})$ on a les implications suivantes :

$$g(n_{21}) > g(n_{22}) \Rightarrow f^{-1}(n_{21}) > f^{-1}(n_{22})$$
$$\Rightarrow f(f^{-1}(n_{21})) > f(f^{-1}(n_{22}))$$

car f conserve l'ordre. Donc : $n_{21} > n_{22}$, ce qui est en contradiction avec l'hypothèse : g conserve l'ordre.

- g (noeud-c) = noeud-c
  g (noeud-b) = noeud-b
  g (comp-b) = comp-b

  En effet, si $n_2 \in$ L'2 est un noeud-c (resp. noeud-b, comp-b), du fait que f est injective et surjective, il existe $n_1 \in$ L'1 tel que n_1 est un noeud-c (resp. noeud-b, comp-b) et tel que $f(n_1) = n_2$ avec $n_2 = n_1$.
  Donc $g(f(n_1)) = g(n_2)$
  D'où : $g(n_2) = n_1$ avec $n_1 = n_2$.

2ème Cas. f n'est pas injective.

Pour montrer que dans ce cas la relation d'équivalence est aussi symétrique, on se ramène au premier cas de la façon suivante :

Si f n'est pas injective, ceci signifie qu'il existe k noeuds ($k > 1$), $n_{11}, n_{12} \ldots, n_{1k} \in$ L'1 tels que $n_{11} < n_{12} < \ldots < n_{1k}$ et tels que $f(n_{11}) = f(n_{12}) = \ldots = f(n_{1k}) = n_{2r} \in$ L'2.

Puisque $n_{11} < n_{12} < \ldots < n_{1k}$ et que f conserve l'ordre on a donc :

$$n_{2r} < n_{2r} < \ldots < n_{2r}$$

Ceci signifie que l'élément n_{2r} est rencontré k fois pendant le parcours de m2. On peut alors définir un ensemble L"2 qui a un élément pour chaque élément de m2 rencontré une fois et k éléments pour chaque élément de m2 rencontré k fois. Soit $n_{2r}^{(i)}$ l'élément de L"2 qui identifie la i-ème rencontre de n_{2r} lors du parcours. Cet ensemble L"2 est muni de la relation d'ordre définie par le parcours de m2, comme l'ensemble L'2.

Soit maintenant la fonction f' : L'1 → L"2 telle que :

- si $n_1 \in$ L'1 n'appartient pas à $\{n_{11}, \ldots, n_{1k}\}$
  alors $f'(n_1) = f(n_1)$
- si $n_1 \in$ L'1 et appartient à $\{n_{11}, \ldots n_{1k}\}$
  alors $f'(n_1) = f'(n_{1i}) = n_{2r}^{(i)}$ pour un certain i.

La fonction f' est injective, elle a les mêmes propriétés que f et on peut donc revenir au premier cas.

On peut noter que si f n'est pas injective, ceci signifie que m1 et m2 ont des représentations non identiques, et pour comparer m1 avec m2 on doit faire plusieurs parcours de certains cheminements de m2. Pour des modes tels que $\underline{m} = \varphi(\underline{m})$, la définition de L"2 correspond à une expansion de ces modes [KO] [KN].

3 - TRANSITIVE : m1 ≙ m2 et m2 ≙ m3 ⇒ m1 ≙ m3.

Si m1 ≙ m2, il existe une fonction f : L'1 → L'2 et m2 ≙ m3, il existe une fonction g : L'2 → L'3. Les fonctions f et g sont toutes les deux surjectives, conservent l'ordre et les égalités.

De plus, on peut supposer que f et g sont injectives, car si elles ne le sont pas, on peut se ramener au cas où elles le sont en effectuant la transformation indiquée dans le deuxième cas ci-dessus.

Soit maintenant une fonction h : L'1 → L'3 telle que h = gof.

- Si $n \in$ L'1, alors on a :
  $h(n) = gof(n) = g(f(n)) \in$ L'3 car $f(n) \in$ L'2
- Si $n_1, n_2 \in$ L'1 et $n_1 < n_2$, alors on a :
  $f(n_1) < f(n_2) \Rightarrow g(f(n_1)) < g(f(n_2)) \Rightarrow h(n_1) < h(n_2)$
- Si $n_1 \in$ L'1 est un noeud-c (resp. noeud-b, comp-b), alors on a :
  $h(n_1) = g(f(n_1)) = g(n_2)$ avec $n_2 \in$ L'2 où n_2 est un noeud-c (resp. noeud-b, comp-b) et $n_2 = n_1$.

$g(n_2) = n_3$ avec $n_3 \in L'3$ où n_3 est un noeud-c (resp. noeud-b, comp-b) et $n_3 = n_2$.

Donc, $h(n_1) = n_3$ avec $n_3 \in L'3$ où n_3 est un noeud-c (resp. noeud-b, comp-b) et $n_3 = n_1$.

- $h(L'1) = g(f(L'1)) = g(L'2) = L'3$.

Donc h est surjective.

La fonction h ainsi définie établit donc l'équivalence $\underline{m1} \doteq \underline{m3}$.

V.4.2 - Démonstration de l'unicité de la représentation minimale -

Soient $p_1(\underline{m})$ et $p_2(\underline{m})$ deux représentations minimales de $\underline{m}$. On veut prouver que $p_1(\underline{m}) \equiv p_2(\underline{m})$.

D'après l'hypothèse, on a :

- $p_1(\underline{m}) \subset \underline{m}$, $\underline{m} \doteq p_1(\underline{m})$ et $p_1(\underline{m})$ est irréductible ;
- $p_2(\underline{m}) \subset \underline{m}$, $\underline{m} \doteq p_2(\underline{m})$ et $p_2(\underline{m})$ est irréductible.

D'autre part, la relation d'équivalence étant symétrique et transitive, on a :

$$p_1(\underline{m}) \doteq p_2(\underline{m})$$

Donc, si L'1 et L'2 sont les ensembles d'éléments de $p_1(\underline{m})$ et de $p_2(\underline{m})$, respectivement, ordonnés d'après les parcours, il existe une correspondance f : L'1 → L'2 qui est surjective et qui conserve l'ordre et l'égalité des noeuds-c, noeuds-b et comp-b.

Si on suppose maintenant que deux représentations minimales peuvent ne pas être identiques (c'est-à-dire que l'une d'elles serait plus synthétique que l'autre) on a les conséquences suivantes :

- $p_1(\underline{m}) \not\equiv p_2(\underline{m}) \Rightarrow$ f n'est pas bijective, d'où :
- soit il n'existe pas f1 : L'1 → L'2, telle que f1 soit injective,

- soit il n'existe pas f2 : L'2 → L'1, telle que f2 soit injective.

Supposons d'abord que f1 n'est pas injective. Il suffit pour cela qu'il existe au moins deux éléments n_{11} et n_{12} de L'1 tels que :

$$n_{11} < n_{12} \quad \text{et} \quad f(n_{11}) = f(n_{12})$$

Dans ce cas, $p_2(\underline{m})$ serait donc plus synthétique que $p_2(\underline{m})$.

Soit maintenant n_{11_s} le successeur immédiat de n_{11} et n_{12_s} le successeur immédiat de n_{12}.

On peut voir que $f(n_{11_s}) = f(n_{11})_s$

et $f(n_{12_s}) = f(n_{12})_s$

En effet, d'une façon générale, si $\underline{m1} \doteq \underline{m2}$ et si M'1 et M'2 sont les ensembles ordonnés par le parcours des éléments de $\underline{m1}$ et de $\underline{m2}$ respectivement, avec la fonction f : M'1 → M'2, pour tout $n \in$ M'1 on a : $f(n_s) = f(n)_s$ où n_s représente le successeur immédiat de n, car, si $f(n_s)$ n'était pas le successeur immédiat de f(n), on aurait l'une des deux possibilités suivantes :

- soit $f(n_s)$ précéde f(n), c'ést-à-dire $f(n_s) < f(n)$, ce qui signifierait que $n_s < n$, car la relation d'équivalence est symétrique et f conserve l'ordre. Du fait que $n < n_s$, $f(n_s)$ ne précéde pas f(n).
- soit $f(n_s)$ suit f(n), mais n'en est pas le successeur immédiat.

  Il existerait alors y = f(x) tel que $f(n) < y < f(n_s)$, ce qui signifie, toujours à cause de la symétrie de l'équivalence et de la conservation de l'ordre que l'on aurait $n < x < n_s$: n_s ne serait donc plus le successeur immédiat de n, ce qui est une contradiction avec l'hypothèse.

Donc, ici, du fait que $f(n_{11}) = f(n_{12})$, on a $f(n_{11_s}) = f(n_{12_s})$.

Les successeurs de $f(n_{11})$ et de $f(n_{12})$ sont donc identiques. De la même façon, on peut considérer les successeurs des successeurs de n_{11} et de n_{12}, etc.... Soit donc N'1 l'ensemble des éléments de L'1 que l'on parcourt si et seulement si l'on passe par n_{11}, et N'2 l'ensemble des éléments de L'1 que l'on parcourt si et seulement si l'on passe par n_{12}.

Il est clair, d'après ce qu'on vient de dire, que f(N'1) = f(N'2).

D'autre part, dans la représentation de $\underline{m1}$, on peut changer toute référence à n_{12} par une référence à n_{11}, toute référence à n_{12_s} par une référence à n_{11_s}, toute référence à $n_{12_{s_s}}$ par une référence à $n_{11_{s_s}}$, etc....

Il reste ainsi un ensemble d'éléments ordonné par le parcours :

$$L"1 = L'1 - \{n_{12}\} - N'2$$

Cet ensemble correspond à la représentation d'un mode que l'on **peut** parcourir, et ce mode est une partie de $p_1(\underline{m})$, car $L"1 \subset L'1$. Appelons $p'_1(\underline{m})$ cette partie de $p_1(\underline{m})$.

Il existe alors une correspondance f' : $L"1 \to L'2$, telle que $f'(n) = f(n)$ pour tout $n \in L"1$. Cette correspondance est surjective, car $f'(L"1) = f(L'1) = L'2$ d'après la relation f(N'1) = f(N'2), et elle conserve l'ordre et les égalités de la même façon que f.

Donc, d'après les définitions de L"1 et de f', on a :

$$p'_1(\underline{m}) \doteq p_2(\underline{m})$$

or puisque : $p_1(\underline{m}) \doteq p_2(\underline{m})$, on a :

$$p'_1(\underline{m}) \doteq p_1(\underline{m})$$

Donc, puisque $p'_1(\underline{m}) \subset p_1(\underline{m})$, $p_1(\underline{m})$ est réductible, ce qui est en contradiction avec l'hypothèse. Cette contradiction vient du fait que f_1 ne serait pas injective. Donc, f_1 est injective. Un raisonnement similaire amène à conclure que f_2 est injective. Donc, f est bijective, ce qui implique que $p_1(\underline{m}) \equiv p_2(\underline{m})$.

Toutes les parties irréductibles d'un mode $\underline{m}$, qui sont équivalentes à ce mode $\underline{m}$, sont donc identiques. Il suffit donc de trouver l'une d'elles pour obtenir la représentation minimale de ce mode.

CHAPITRE 3

L E S M O D I F I C A T I O N S

Dans ce qui suit on s'intéresse aux problèmes des modifications en BASEL, c'est-à-dire aux transformations d'une valeur de mode n en une valeur de mode m, lorsque le contexte l'impose. On présentera quelques notions de base et la définition de la compatibilité de deux modes afin de définir l'algorithme de modification utilisé par le compilateur. Enfin on présentera toutes les possibilités d'appel de cet algorithme au cours de la compilation d'un programme BASEL.

I - NOTIONS DE BASE ET NOTATIONS -

1 - On parlera indifféremment de modification et de transformation bien que ce dernier terme soit plutôt employé dans le cas où un mode n devient mode m, et le premier dans le cas où on s'intéresse seulement au changement de n sans préciser en quel mode.

2 - Une valeur de mode n sera notée v_n. Une transformation de v_n en v_m sera notée $t(v_n) = v_m$, et on notera aussi $t(n) = m$ la transformation de mode.
$t(v_n) = v_m$ implique toujours $t(n) = m$ et réciproquement. Parfois on omettra l'une de deux notations.

3 - Les modifications en BASEL.

On distingue les cas suivants :

3a - Transformations élémentaires : ce sont celles définies dans la présentation du langage (voir Ch.1, VII) :

Transformation		n	m = t(n)	description $v_m = t(v_n)$
notation	nom			
DR	dérepérage	ptr n1	n1	la valeur repérée par l'adresse v_n.
AP	appel	proc n1	n1	le résultat de la procédure sans paramètres v_n.
EX	extension	n ≠ union [...] ou n = union [n1, ..., np]	union [..., n, ...] ou union [..., n1, ..., np, ...]	une valeur de mode union, dont n est un des modes possibles ou une valeur de mode union, dont les ni sont des modes possibles.
SQ	séquentialisation	tuple [n1, ..., n1] ou struct [n1 s1, ..., n1 sp]	seq n1	la séquence dont les éléments sont ceux de la tuple ou de la structure.
ST	structuration	tuple [n1, ..., np]	struct [n1 s1, ..., np sp]	la structure dont les valeurs des champs sont celles des éléments de la tuple.
DS	déstructuration	struct [n1 s1, ..., np sp]	tuple [n1, ..., np]	la tuple dont les éléments sont les valeurs des champs de la structure.

SP	suppression	ptr* int ou ptr* real ou ptr* bool ou ptr* char ou ptr* seq n1 ou ptr* tuple [n1, ..., np] ou ptr* struct [n1 s1, ..., np sp]ou ptr* union [n1, ..., np] ou ptr* proc [n1, ..., np] N	none	absence de valeur

On appellera ξ l'ensemble {DR, AP, EX, SQ, ST, DS, SP}

3b - Composition des transformations élémentaires :

Soient $t_1, t_2, \ldots, t_k \in \xi$ telles que :

$$t_k (v_n) = v_{n(k-1)}$$

$$t_{k-1}(v_{n(k-1)}) = v_{n(k-2)}$$

$$\vdots$$

$$t_1 (v_{n1}) = v_m$$

On appellera t la transformation définie à partir de la composition des t_i, $i \in [1, k]$:

$$t(v_n) = t_1\ t_2 \ldots t_k\ (v_n) = t_1\ (t_2\ (\ldots\ (t_k(v_n))\ \ldots)) = v_m$$

Exemple :

SQ DR (ptr tuple [int, int]) = seq int

Il est évident qu'il n'est pas possible de composer t_i avec t_j pour tout $(t_i, t_j) \in \mathcal{E} \times \mathcal{E}$. Par exemple, la composition DP SQ n'existe pas, car SQ (v_n) est toujours une valeur de mode seq m, qu'il est impossible de dérepérer.

3c - Ensembles des transformations -

Soit l'ensemble ordonné $\{t_1, t_2, \ldots, t_p\}$ où t_i est une transformation élémentaire ou une composition de transformations élémentaires pour $i \in [1, p]$;

soit n = tuple [n1, ..., np] ou n = struct [n1 s1, ..., np sp] ;

l'application de l'ensemble des transformations à v_n est une valeur dont le mode est défini par :

$\{t_1, t_2, \ldots, t_p\}$ (n) = tuple $[t_1(n1), t_2(n2), \ldots, t_p(np)]$ si n = tuple [n1, n2, ..., np]
et par
$\{t_1, t_2, \ldots, tp\}$ (n) = struct $[t_1(n1)$ s1, $t_2(n2)$ s2, ..., $t_p(np)$ sp] si n = struct = [n1 s1, n2 s2, ..., np sp] '

C'est-à-dire : la transformée d'un ensemble est l'ensemble de transformées des valeurs qui le composent.

Exemple :

{DR AP, DR} (struct [proc ptr bool b, ptr int i]) = struct [bool b, int i]

3d - Transformations -

Les cas définis dans 3a, 3b et 3c peuvent se combiner entre-eux, définissant ainsi des nouvelles transformations. Par conséquent, si l'on ajoute la transformation I telle que $I(v_n) = v_n$ pour tout mode n,

l'ensemble des transformations en BASEL est défini par :

$\mathcal{T}$ = {t tels que t = I ou t $\in \mathcal{E}$ ou t = $t_1\ t_2 \ldots t_k$ avec $t_i \in \mathcal{T}$, $i \in [1, k]$ ou t = $\{t_1, t_2, \ldots, t_p\}$ avec $t_i \in \mathcal{T}$, $i \in [1, p]$}.

Exemple :

n = ptr proc tuple [struct [ptr int s], tuple [int], proc tuple [ptr ptr int]]

t = SQ {DS {DP}, I, {DR DR} AP} AP DR

t (n) = seq tuple [int]

II - COMPATIBILITE DE DEUX MODES -

II.1 - Fonction de compatibilité : f(m, n)

On définit la compatibilité de deux modes au moyen d'une fonction f : $\mathcal{M} \times \mathcal{M} \rightarrow$ {vrai, faux}, où $\mathcal{M}$ est l'ensemble de tous les modes possibles en BASEL.

Définition : n $\wp$ m (n est compatible avec m) si et seulement si f(m, n) = vrai.
La fonction f(m, n) est définie de façon récursive à l'aide de la matrice ci-dessous, où les lignes représentent les m et les colonnes les n.
n n'est pas compatible avec m est noté n $\not\wp$ m.

DEFINITION DE f : $\mathcal{M} \times \mathcal{M} \to$ {vrai, faux}

m \ n	mb : int	mb : real	mb : bool	mb : char	m1 : ptr n1	m1 : seq n1	m1 : tuple [n1, ..., nq]	m1 : struct [n1 s1, ..., nq sq]	mu : union [n1, ..., nq]	m1 : proc [n1, ..., nq] N	m1 : proc n1
mb : int	(*) V	(**) F	F	F	f(m, n1)	F	F	F	F	F	f(m, n1)
mb : real	F	V	F	F	f(m, n1)	F	F	F	F	F	f(m, n1)
mb : bool	F	F	V	F	f(m, n1)	F	F	F	F	F	f(m, n1)
mb : char	F	F	F	V	f(m, n1)	F	F	F	F	F	f(m, n1)
m1 : ptr m1	F	F	F	F	$(m = n) \cup f(m, n1)$	F	F	F	F	F	f(m, n1)
m1 : seq m1	F	F	F	F	f(m, n1)	$(m = n)$	$\bigcap_{i=1}^{q} f(m1, ni)$	$\bigcap_{i=1}^{q} f(m1, ni)$	F	F	f(m, n1)
m1 : tuple [m1, ..., mp]	F	F	F	F	f(m, n1)	F	$(m = n) \cup [(p=q) \cap \bigcap_{i=1}^{p} f(mi, ni)]$	$(p = q) \cap \bigcap_{i=1}^{p} f(mi, ni)$	F	F	f(m, n1)
m1 : struct [m1 s1, ..., mp sp]	F	F	F	F	f(m, n1)	F	$(p=q) \cap \bigcap_{i=1}^{p} f(mi, ni)$	$(m = n) \cup [(p=q) \cap \bigcap_{i=1}^{p} f(mi, ni)]$	F	F	f(m, n1)
mu : union [m1, ..., mp]	(***) $comp_m(n)$	$comp_m(n)$	$comp_m(n)$	$comp_m(n)$	$comp_m(n)$	$comp_m(n)$	$comp_m(n)$	$comp_m(n)$	$(m = n) \cup \bigcap_{i=1}^{q} comp_m(ni)$	$comp_m(n)$	$comp_m(n)$
m1 : proc [m1, ..., mp]M	F	F	F	F	f(m, n1)	F	F	F	F	$(m = n)$	f(m, n1)
m1 : proc m1	F	F	F	F	f(m, n1)	F	F	F	F	F	$(m = n) \cup f(m, n1)$

(*) V représente la valeur vrai

(**) F représente la valeur faux

(***) $comp_m(n)$ est vrai (représenté par comp(n) = 1) si et seulement si le mode n est un des modes qui composent m (Voir Ch.2, I.2.2)

Le problème de la vérification des modes au cours de la compilation d'un programme BASEL pose le problème de l'existence et de la définition de t, tel que t(n) = m quand on a une valeur de mode n, et m est le mode imposé par le contexte.

On démontrera par la suite que la compatibilité de n avec m est une condition nécessaire et suffisante pour l'existence et la détermination d'un tel t. Auparavant, il est nécessaire de faire quelques remarques sur le calcul de f(m, n).

II.2 - Niveau de récursivité de f(m, n) -

Le calcul de f(m, n) pour m et n donnés, peut impliquer le calcul d'une autre valeur de la fonction (f(m, n1), f(m1, ni) ou f(mi, ni) i $\in$ [1, p]) d'après la définition récursive de f.

On définit le niveau de récursivité r de f(m, n) que l'on note r(f(m, n)) de la façon suivante :

r(f(m, n)) = 0 si f(m, n) = V ou f(m, n) = F sans qu'il soit nécessaire de calculer une autre valeur de f.

r(f(m, n)) = 1 si le calcul de f(m, n) implique le calcul de :

- f(m, n1) avec r(f(m, n1)) = 0 ou de
- f(m1, ni) i $\notin$ [1, p] avec r(f(m1, ni)) = 0 pour tout i $\in$ [1, p] ou de
- f(mi, ni), i $\in$ [1, p] avec r(f(mi, ni)) = 0 pour tout i $\in$ [1, p],

c'est-à-dire si le calcul de f(m, n) implique un autre calcul, celui de

f(m, n1) ou de $\bigcap_{i=1}^{p}$ f(m1, ni) ou de $\bigcap_{i=1}^{p}$ f(mi, ni).

En général, on parlera de niveau de récursivité égal à k (k $\geq$ 1) : r(f(m, n)) = k si pour calculer f(m, n) il faut calculer :

- $f(\underline{m}, \underline{n1})$ avec $r(f(\underline{m}, \underline{n1})) = k-1$, ou

- $\bigcap_{i=1}^{p} (f(\underline{m1}, \underline{ni})$ avec $(\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})) = k-1$, ou

- $\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})$ avec $(\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})) = k-1$.

Le niveau de récursivité de $\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})$ (respectivement de $\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})$) est défini par :

$$r(\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})) = \max_{i} r(f(\underline{m1}, \underline{ni}))$$

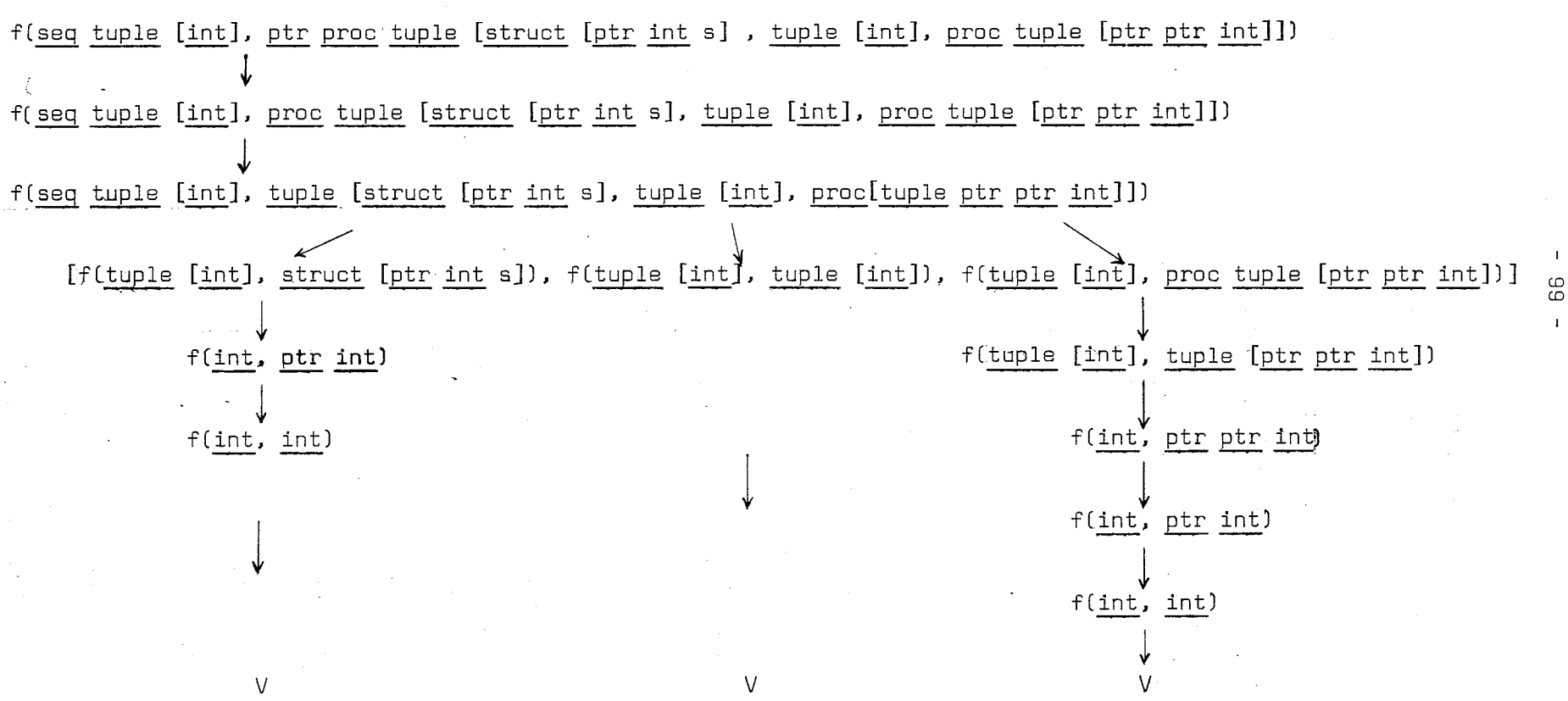
(et par $\max_{i} r(f(\underline{mi}, \underline{ni}))$ respectivement).

Exemple :

n = ptr proc tuple [struct [ptr int s],tuple [int], proc tuple [ptr ptr int]]

m = seq tuple [int]

le calcul de f(m, n) se fait d'après le schéma suivant :



$r(f(m, n)) = 7$; $r(\bigcap_{i=1}^{3} f(m1, ni)) = 4$

Au moyen de la notion de niveau de récursivité, on peut alors démontrer l'existence et l'unicité de t.

II.3 - $f(\underline{m}, \underline{n}) = V \Leftrightarrow$ il existe un et un seul $t \in \mathcal{T}$ tel que $t(\underline{n}) = \underline{m}$.

($\Rightarrow$) on la démontre par induction sur le niveau de récursivité r.

DEMONSTRATION DE $f(m,n) = V \Rightarrow$ il existe un et un seul $t \in \mathcal{T}$ tel que $t(n) = m$

1er Cas : r = 0

Si r = 0 et $f(\underline{m}, \underline{n}) = V$ alors deux cas peuvent se présenter :

- soit $\underline{m} = \underline{n}$ ce qui implique t = I
- soit $\underline{m} = \underline{union}$ [... $\underline{n}$ ...] ou $\underline{m} = \underline{union}$ [... $\underline{n1}$, $\underline{n2}$, ..., $\underline{np}$, ...] ce qui implique t = EX.

2ème Cas : r = 1

Si r = 1 et $f(\underline{m}, \underline{n}) = V$ alors l'inspection de la matrice de définition de f montre que un et un seul des cas suivants peut se présenter :

a) - soit $f(\underline{m}, \underline{n})$ implique le calcul de $f(\underline{m}, \underline{n1})$ avec $r(f(\underline{m}, \underline{n1})) = 0$, alors $\underline{n1} = DR(\underline{n})$ et $\underline{m} = \underline{n1}$, donc $\underline{m} = I\ DP(\underline{n}) = DP(\underline{n})$ ou
$\underline{n1} = AP(\underline{n})$ et $\underline{m} = \underline{n1}$, donc $\underline{m} = I'AP(\underline{n}) = AP(\underline{n})$

Il est à noter que $\underline{m} = EX(\underline{n1})$ ne se présente pas car les seules cas où on est obligé de calculer $f(\underline{m}, \underline{n1})$ excluent cette possibilité.

b) - soit $f(\underline{m}, \underline{n})$ implique le calcul de $\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})$ avec

$r(\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})) = 0$

alors $f(\underline{m1}, \underline{ni}) = f(\underline{m1}, \underline{m1})$ donc $\underline{m1} = I(\underline{ni})$ pour un i, ou
$f(\underline{m1}, \underline{ni}) = f(\underline{union}\ [..., \underline{ni}, ...], \underline{ni})$ donc $\underline{m1} = EX(\underline{ni})$ pour un i, ou
$f(\underline{m1}, \underline{ni}) = f(\underline{union}\ [..., \underline{ni1}, ..., \underline{nip_i}, ...], \underline{union}\ [\underline{ni1}, ..., \underline{nip_i}])$

donc $\underline{m1} = EX(\underline{ni})$ pour un i

donc $\underline{m} = SQ\ \{t_1, t_2, ..., t_p\}\ (\underline{n})$, avec $t_i = I$ ou $t_i = EX$, $i \in [1, p]$

c) soit $f(\underline{m}, \underline{n})$ implique le calcul de $\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})$

avec $r(\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})) = 0$

alors, d'une façon analogue a b) :

$f(\underline{mi}, \underline{ni}) = I(\underline{ni})$ ou

$f(\underline{mi}, \underline{ni}) = EX(\underline{ni})$, $i \in [1, p]$

donc $\underline{m} = DS\ \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$ ou

$\underline{m} = ST\ \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$ ou

$\underline{m} = \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$, avec $t_i = I$ ou $t_i = EX$, $i \in [1, p]$

En particulier :

si $\underline{m}$ = tuple [...] et $\underline{n}$ = tuple [...] ou

$\underline{m}$ = struct [...] et $\underline{m}$ = struct [...]

et

$t_i = I$ pour tout $i \in [1, p]$

alors $\underline{m} = I(\underline{n}) = \underline{n}$, c'est le cas r = 0.

3ème Cas : r = k, k > 1

On admet que pour tout $\underline{m}$, $\underline{n}$ tels que $r(f(\underline{m}, \underline{n})) \leq k$, il existe un et un seul $t \in \mathcal{T}$ tel que $\underline{m} = t(\underline{n})$.

Soit $f(\underline{m}, \underline{n})$ avec $r(f(\underline{m}, \underline{n})) = k+1$.

Si $r(f(\underline{m}, \underline{n})) = k+1$, alors le calcul de $f(\underline{m}, \underline{n})$ détermine un des cas suivants :

a) - soit calculer $f(\underline{m}, \underline{n1})$ avec $r(f(\underline{m}, \underline{n1})) = k$, donc $\underline{m} = t(\underline{n1})$ par hypothèse, mais $\underline{n1} = DR(\underline{n})$ ou $\underline{n1} = AP(\underline{n})$,

donc $m = t(DR(\underline{n}))$ ou $\underline{m} = t(AP(\underline{n}))$

c'est-à-dire $\underline{m} = t'(\underline{n})$

b) - soit calculer $\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})$ avec $r(\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})) = k$

donc $\underline{n}$ = tuple [$\underline{n1}$, $\underline{n2}$, ..., $\underline{np}$] ou $\underline{n}$ = struct [$\underline{n1}$ s1, $\underline{n2}$ s2, ..., $\underline{np}$ sp].

Etant donné que pour chaque $i \in [1, p]$: $\underline{m1} = t_i(\underline{ni})$, par hypothèse, il est possible de définir :

$\underline{n'}$ = **tuple** [$\underline{m1}$, ..., $\underline{m1}$] si $\underline{n}$ = **tuple** [$\underline{n1}$, $\underline{n2}$, ..., $\underline{np}$] ou

$\underline{n'}$ = **struct** [$\underline{m1}$ s1, ..., $\underline{m1}$ sp] si $\underline{n}$ = **struct** [$\underline{n1}$ s1, $\underline{n2}$, ..., $\underline{np}$ sp]

et $\underline{n'} = \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$

mais d'après le deuxième cas :

$$\underline{m} = SQ\ \{I, I, \ldots, I\}\ (\underline{n'}) = SQ\ \{I, I, \ldots, I\}\ \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$$

c'est-à-dire : $\underline{m} = SQ\ \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$

c) - soit calculer $\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})$ avec $r(\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})) = k$

donc $\underline{n}$ = **tuple** [$\underline{n1}$, $\underline{n2}$, ..., $\underline{np}$] ou $\underline{n}$ = **struct** [$\underline{n1}$ s1, $\underline{n2}$ s2, ..., $\underline{np}$ sp].

Etant donné que pour chaque $i \in [1, p]$: $\underline{mi} = t_i(\underline{ni})$ par hypothèse, il est possible de définir :

$\underline{n'}$ = **tuple** [$\underline{m1}$, $\underline{m2}$, ..., $\underline{mp}$] ou

$\underline{n'}$ = **struct** [$\underline{m1}$ s1, $\underline{m2}$ s2, ..., $\underline{mp}$ sp]

et $\underline{n'} = \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$

mais d'après le premier ou deuxième cas :

$\underline{m}$ = DS {I, I, ..., I} ($\underline{n'}$) ou

$\underline{m}$ = ST {I, I, ..., I} ($\underline{n'}$) ou

$\underline{m}$ = I {I, ..., I} ($\underline{n'}$)

or $\underline{m} = X\ \{I, I, \ldots, I\}\ (\underline{n'}) = X\ \{I, I, \ldots, I\}\ \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$

c'est-à-dire : $\underline{m} = X\ \{t_1, t_2, \ldots, t_p\}\ (\underline{n})$

avec : X = DS ou X = ST ou X = I

Remarque sur SP.

Il est à noter que dans la définition de t, SP n'intervient jamais. Cela est dû au fait que la suppression s'applique à des valeurs v_n, avec $n \not\in$ none. Les modes compatibles avec none sont ceux qui sont compatibles avec :

ptr* int ou avec
ptr* real ou avec
ptr* bool ou avec
ptr* char ou avec
ptr* seq n1 ou avec
ptr* tuple [n1, ..., np] ou avec
ptr* struct [n1 s1, ..., np sp] ou avec
ptr* union [n1, ..., np] ou avec
ptr* proc [n1, ..., np]N

Pour supprimer une valeur, il est donc nécessaire d'étudier la compatibilité avec ces modes, en faisant si nécessaire des transformations, pour appliquer en fin SP. Les modifications possibles sont de la forme :

$$t = (DR^* \; AP)^*$$

Unicité de t.

Dans la matrice de définition de f(m , n), les calculs de niveau 0 sont définis de manière unique. On vient de démontrer que tout calcul de niveau de récursivité k se ramène de manière unique a un calcul de niveau k-1 pour tout $k \geq 1$, la transformation t définie par ce calcul est donc toujours unique.

Il est intéressant de noter que $n \not\in m$ pour n = ptr int et m = union [int], d'après la définition de f.

Si l'on remplace la définition de l'élément f(m = union [...], n = ptr ...) par :

$$comp_{\underline{m}}(\underline{n}) \cup comp_{\underline{m}}(\underline{n1})$$

le mode ptr int serait compatible à union [int], avec union [int] = EX DR (ptr int).

Alors l'unicité de t ne pourrait plus être assurée, car dans un cas comme le suivant :

$\underline{n}$ = ptr int $\underline{m}$ = union [int, ptr int]

$\underline{n} \not\sim \underline{m}$ et : $\underline{m} = EX(\underline{n})$ ou $\underline{m} = EX\ DR(\underline{n}) \Rightarrow \underline{m} = t_1(\underline{n}) = t_2(\underline{n})$ avec $t_1 \neq t_2$

Le langage interdit cette possibilité.

(⇐) DEMONSTRATION DE il existe $t \in \mathcal{T}$ tel que $t(\underline{n}) = \underline{m} \Rightarrow f(\underline{m}, \underline{n}) =$ vrai.

Il est équivalent de démontrer :

$f(\underline{m}, \underline{n})$ = faux ⇒ il n'existe pas $t \in \mathcal{T}$ tel que $t(\underline{n}) = \underline{m}$

1er Cas : r = 0.

Si $r(f(\underline{m}, \underline{n})) = 0$ et $f(\underline{m}, \underline{n})$ = faux aucune transformation peut s'appliquer à $\underline{n}$ pour obtenir $\underline{m}$, d'après la définition de transformations élémentaires.

2ème Cas : r > 0.

Le calcul de $f(\underline{m}, \underline{n})$ avec $r(f(\underline{m}, \underline{n})) > 0$ détermine toujours d'une façon unique une composition de transformations. $f(\underline{m}, \underline{n})$ = faux implique que dans un certain niveau du calcul un et un seul des cas suivants peut se présenter :

a) soit $f(\underline{m}, \underline{n1})$ = faux avec $r(f(\underline{m}, \underline{n1})) = 0$,

b) soit $\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})$ = faux avec $r(f(\underline{m1}, \underline{ni})) = 0$ pour un i,

c) soit $\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})$ = faux avec $r(f(\underline{mi}, \underline{ni})) = 0$ pour un i,

ce qui fait finir le calcul de $f(\underline{m}, \underline{n})$. La composition de transformations qui se définit au fur et à mesure du calcul de $f(\underline{m}, \underline{n})$ est donc arrêtée sans pouvoir la déterminer entièrement :

a) $f(\underline{m}, \underline{n1})$ = faux $\Rightarrow$ il n'existe pas $t \in \mathcal{T}$ tel que $\underline{m} = t(\underline{n1})$

b) $\bigcap_{i=1}^{p} f(\underline{m1}, \underline{ni})$ faux $\Rightarrow$ il n'existe pas $t \in \mathcal{T}$ tel que $\underline{m1} = t(\underline{ni})$.

pour un i, donc il est impossible de construire l'ensemble $\{t_1, t_2, \ldots, t_p\}$ qui s'applique à $\underline{n}$.

c) $\bigcap_{i=1}^{p} f(\underline{mi}, \underline{ni})$ = faux $\Rightarrow$ il n'existe pas $t \in \mathcal{T}$ tel que $\underline{mi} = t(\underline{ni})$

pour un i, donc il est impossible de construire l'ensemble $\{t_1, t_2, \ldots, t_p\}$ qui s'applique à $\underline{n}$.

Dans tous les trois cas : il est impossible d'exprimer $\underline{m}$ comme une transformation de $\underline{n}$.

D'après la démonstration de la condition nécessaire et suffisante de l'existence de t, on peut conclure que f ainsi définie représente toutes les possibilités de modification admises par le langage.

II.4 - Propriétés de la compatibilité -

La compatibilité est :

1 - REFLEXIVE : $\underline{n} \mathcal{C} \underline{n}$ pour tout $\underline{m}$.

En effet : $\underline{n} = I(\underline{n}) \Rightarrow f(\underline{n}, \underline{n}) =$ vrai.

2 - ANTISYMETRIQUE : $\underline{n} \mathcal{C} \underline{m} \not\Rightarrow \underline{m} \mathcal{C} \underline{n}$.

Un exemple suffit à le prouver :

si $\underline{n}$ = proc int et $\underline{m}$ = int, $\underline{m}$ = AP($\underline{n}$) et f($\underline{m}$, $\underline{n}$) = vrai ; mais f($\underline{n}$, $\underline{m}$) = faux car il n'existe pas $t \in \mathcal{T}$ tel que

$$\underline{\text{proc}}[\underline{\text{int}}] = t(\underline{\text{int}})$$

3 - TRANSITIVE : $\underline{n} \mathcal{C} \underline{p}$ et $\underline{p} \mathcal{C} \underline{m} \Rightarrow \underline{n} \mathcal{C} \underline{m}$

$\underline{n} \mathcal{C} \underline{p} \Rightarrow$ il existe $t_1 \in \mathcal{T}$ unique tel que $\underline{p} = t_1(\underline{n})$,
$\underline{p} \mathcal{C} \underline{m} \Rightarrow$ il existe $t_2 \in \mathcal{T}$ unique tel que $\underline{m} = t_2(\underline{p})$.

Alors : $\underline{m} = t_2(\underline{p}) = t_2(t_1(\underline{n}))$, donc $\underline{m} = t(\underline{n})$ avec t déterminé de façon unique d'après l'unicité de t_1 et de t_2. Du fait que $\underline{m} = t(\underline{n})$: f($\underline{m}$, $\underline{n}$) = vrai $\Rightarrow \underline{n} \mathcal{C} \underline{m}$.

III - ALGORITHME DE MODIFICATION AU COURS DE LA COMPILATION -

Il sera décrit à l'aide d'une fonction procédure : MODIFICATION ($\underline{m}$, $\underline{n}$) qui a comme paramètres :

$\underline{m}$, le mode imposé par le contexte et
$\underline{n}$, le mode de la valeur à modifier.

La procédure établit si $\underline{n} \mathcal{C} \underline{m}$ (résultat vrai) ou $\underline{n} \not\mathcal{C} \underline{m}$ (résultat faux) et modifie le mode $\underline{n}$ en générant les instructions nécessaires pour modifier la valeur $v_{\underline{n}}$ au moment de l'exécution. On doit considérer qu'à la sortie de la procédure $\underline{n}$ est le transformé du mode original $\underline{n}$.

Si $n \mathcal{C} m$ le mode n à la sortie est égal à m ;

Si $n \not\mathcal{C} m$ il est différent de m et éventuellement égal au mode n original : c'est le cas si aucune modification n'a pu être faite.

Dans le premier cas ($n \mathcal{C} m$) le contexte est correct. Dans le deuxième cas : ($n \not\mathcal{C} m$) si un seul mode m est possible, le contexte est incorrect. Mais s'il existe un autre mode m' possible ($m' \neq m$), alors il est nécessaire de faire un nouvel appel de la procédure avec m' comme paramètre.

III.1 - Matrice de modifications $\mathcal{F}$[SC].

La procédure MODIFICATION (m, n) utilise la matrice $\mathcal{F}$ où les lignes représentent les m et les colonnes les n. Les éléments de $\mathcal{F}$ sont des suites d'instructions.

L'appel MODIFICATION (m, n) implique le choix d'un élément de $\mathcal{F}$ et l'exécution des instructions qui le composent. La matrice $\mathcal{F}$ est construite d'après la définition de f(m, n).

Dans cette matrice : t(n) avec $t \in \mathcal{E}$ symbolise la transformation de n et la génération des instructions qui exécuteront $t(v_n)$. La signification des instructions est classique : itération, affectation, instruction, conditionnelle, etc....

Dans cette même matrice, χ symbolise un mode quelconque différent de none tel que $n \not\mathcal{C} \chi$ pour tout $n \in \mathcal{M}$. χ sert à établir la compatibilité dans le cas où l'on veut seulement vérifier un constructeur de mode mais non toute la construction du mode. Par exemple, si l'on veut vérifier que le mode n est compatible avec seq ... (soit seq int, soit seq tuple [bool, char], soit seq p pour tout $p \in \mathcal{M}$, $p \neq$ none), on appelle MODIFICATION (seq χ, n). Ainsi :

seq ptr int $\not\mathcal{C}$ seq χ,
tuple [int, real] $\not\mathcal{C}$ tuple [χ, χ]

Remarque : les différentes occurences de χ dans une expression ne symbolisent pas nécessairement des modes identiques, comme le montre l'exemple ci-dessus.

III.2 - Définition de $\mathcal{F}$.

m = \ n =	int	real	bool	char	ptr n1
int	V	F	F	F	n ← DR(n) ; MODIFICATION(m, n)
real	F	V	F	F	n ← DR(n) ; MODIFICATION(m, n)
bool	F	F	V	F	n ← DR(n) ; MODIFICATION(m, n)
char	F	F	F	V	n ← DR(n) ; MODIFICATION(m, n)
ptr m1	F	F	F	F	si m1 = $\mathcal{X}$ alors V si m = n alors V n ← DR(n) ; MODIFICATION(m, n)
seq m1	F	F	F	F	n ← DR(n) ; MODIFICATION(m, n)
tuple [m1, m2, ..., mp]	F	F	F	F	n ← DR(n) ; MODIFICATION(m, n)
struct [m1 sm1, m2 sm2, ..., mp smp]	F	F	F	F	n ← DR(n) ; MODIFICATION(m, n)
union [m1, m2, ..., mp]	si M-U (m, n) = 0 alors F n ← EX(n) ; V	si M-U (m, n) = 0 alors F n ← EX(n) ; V	si M-U (m, n) = 0 alors F n ← EX(n) ; V	si M-U (m, n) = 0 alors F n ← EX(n) ; V	si M-U(m, n) = 0 alors F n ← EX(n) ; V
proc [m1, m2, ..., mp] M	F	F	F	F	n ← DR(n) MODIFICATION(m, n)
proc m1	F	F	F	F	n ← DR(n) ; MODIFICATION(m, n)
$\mathcal{X}$	V	V	V	V	V

instructions	**signification**
V	sortir de la procédure, le résultat est vrai
F	sortir de la procédure, le résultat est faux
n ← t(n)	avec t ∈ $\mathcal{T}$: instruction d'affectation. Elle symbolise la génération du code de transformation de la valeur
si ... alors si ... alors ... sinon	instruction conditionnelle
pour i(1 : p) :	instruction itérative
Remarque : le symbole ; est un séparateur d'instructions.	

seq n1	tuple [n1, n2, ..., nq]	struct [n1 sn1, n2 sn2, ..., nq snq]	union [n1, ..., nq]	proc [n1, ..., nq] N	proc n1
F	F	F	F	F	n ← AP(n) ; MODIFICATION(m, n)
F	F	F	F	F	n ← AP(n) ; MODIFICATION(m, n)
F	F	F	F	F	n ← AP(n) ; MODIFICATION(m, n)
F	F	F	F	F	n ← AP(n) ; MODIFICATION(m, n)
F	F	F	F	F	n ← AP(n) ; MODIFICATION(m, n)
si m1 = $\mathcal{X}$ alors V m = n	si m1 = $\mathcal{X}$ alors F pour i(1 : q) : si ¬MODIFICATION(m1,ni) alorsF n ← SQ(n) ; V	si m1 = $\mathcal{X}$ alors F pour i(1 : q) : si ¬MODIFICATION(m1,ni) alors F n ← SQ(n) ; V	F	F	n ← AP(n) ; MODIFICATION(m, n)
F	si m = n alors V si p ≠ q alors F pour i(1 : p) : si ¬MODIFICATION(mi,ni)alors F V	si p ≠ q alors F pour i(1 : p) : si ¬MODIFICATION(mi,ni) alors F n ← OS(n) ; V	F	F	n ← AP(n) ; MODIFICATION(m, n)
F	si p ≠ q alors F pour i(1 : p) : si ¬MODIFICATION(mi,ni)alors F n ← ST(n) ; V	si m = n alors V si p ≠ q alors F pour i(1 : p) : si MODIFICATION(mi,ni) ou (smi ≠ sni) alors F V	F	F	n ← AP(n) ; MODIFICATION(m, n)
si M-U(m, n) = 0 alors F n ← EX(n) ; V	si M-U(m, n) = 0 alors F n ← EX(n) ; V	si M-U(m, n) = 0 alors F n ← EX(n) ; V	si m = n alors V pour i(1 : q) : si M-U(m, ni) = 0 alors F n ← EX(n) ; V	si M-U(m, n) = 0 alors F n ← EX(n) ; V	si M-U(m, n) = 0 alors F n ← EX(n) ; V
F	F	F	F	si m1 = $\mathcal{X}$ alors V si m = n alors V sinon F	n ← AP(n) ; MODIFICATION(m, n)
F	F	F	F	F	si m1 = $\mathcal{X}$ alors V si m = n alors V n ← AP(n) ; MODIFICATION(m, n)
V	V	V	V	V	V

IV - VERIFICATION DES MODES AU COURS DE LA COMPILATION -

Dans ce qui suit on décrit la vérification de modes au cours de la compilation d'un programme BASEL, c'est-à-dire : toutes les possibilités d'appel de MODIFICATION (m, n). Chaque appel de la procédure est présenté de la façon informelle suivante :

si MODIFICATION (m, n) alors ... sinon ... ;

On utilise B pour indiquer que le contexte est correct et E pour signaler qu'il ne l'est pas.

Avant d'aborder les différents appels, quelques remarques sur la modification des modes sont nécessaires. Elles vont permettre d'en tirer des conclusions sur la vérification au cours de la compilation.

IV.1 - Forme de t avec t(n) = m pour certains m $\in \mathcal{M}$.

1 - Tout mode n compatible avec un mode de base (mb) est de la forme n = (ptr* proc*)* mb, car d'après les quatre premières lignes de $\mathcal{F}$, seules les transformations DR et AP permettent d'obtenir un mb.

2 - a) Tout mode n compatible avec tuple [χ, χ, ..., χ] avec p occurrences de χ est de la forme :

n = (ptr* proc*)* tuple [n1, n2, ..., np] ou
n = (ptr* proc*)* struct [n1 s1, n2 s2, ..., np sp]

car d'après la ligne correspondante de $\mathcal{F}$, il n'y a que les transformations DR et AP et éventuellement DS qui permettent d'obtenir une tuple.

b) Tout mode n compatible avec struct [χ s1, ..., χ sp] est de la forme :

n = (ptr* proc*)* struct [n1 s1, ..., np sp] ou
n = (ptr* proc*)* tuple [n1, ..., np]

Les seules transformations sont DR, AP et éventuellement ST.

3 - Tout mode n compatible avec seq χ est de la forme :

n = (ptr* proc*)* seq n1

puisqu'on ne peut faire que des DR et/ou AP.

En effet, SQ est exclue par définition quand on cherche une séquence sans préciser le mode de ses éléments, afin d'éliminer des ambiguïtés, comme le montre l'exemple ci-dessous :

si n = tuple [tuple [int], struct [int s], tuple [ptr int]] et
m = seq χ , alors

t(n) = SQ {ST, I, ST {DR}} (n) = seq struct [int s] et
t'(n) = SQ {I, DS, {DR}} (n) = seq tuple [int]

seraient corrects tous les deux.

4 - a) Tout mode n compatible avec proc χ est de la forme :

n = ptr* proc n1

La seule modification possible est DR.

b) Tout mode n compatible avec ptr χ est de la forme :

n = proc* ptr n1

La seule modification à appliquer est donc AP.

5 - Tout mode n compatible avec proc [χ]χ est de la forme :

n = (ptr* proc*)* proc [n1, ..., np]N, p $\geqslant$ 1.

d'après la définition de $\mathcal{F}$. Les seules transformations possibles sont donc DR et AP.

IV.2 - Vérification des modes dans certains contextes particuliers -

D'après ce qui vient d'être dit en IV.1, on peut en déduire les conséquences suivantes :

1 - Si le contexte est tel que m1 et m2 sont des modes mb possibles, alors :

si MODIFICATION (m1, n) = vrai le contexte est correct.

Mais si MODIFICATION (m1, n) = faux, n a été transformé en un mode différent de m1. Si ce mode est égal à m2 le contexte est correct. Les modifications que l'on a fait à n sont celles que l'on aurait fait si on avait exécuté directement MODIFICATION (m2, n) puisqu'on a pu enlever que les ptr et/ou proc. Donc, si MODIFICATION (m1, n) = faux il ne faut pas appeler MODIFICATION (m2, n) ni supprimer les transformations de v_n générées par le premier appel.

Il sufffit de voir si le mode résultant est égal à m2 ; si oui le contexte est bon ; si non, il est incorrect. Dans ce dernier cas, les modifications générées peuvent être supprimées ou non, de toutes façons le programme n'est pas correct.

2 - Un cas analogue se présente si m1 = tuple [int, int] et m2 = tuple [real, real] sont des modes possibles.

Un premier appel MODIFICATION (tuple [λ, χ], n) fait (DR* AP*)* ou DS (DR* AP*)*.

Si l'appel fournit le résultat vrai alors on appelle MODIFICATION (m1, n) où n est le mode résultant des modifications du premier appel.

Le deuxième appel fait (DR* AP*)* pour chaque élément de n. S'il fournit le résultat vrai, le contexte est correct, s'il donne faux, le mode résultant doit être égal à m2 pour que le contexte soit correct.

Le même raisonnement peut se généraliser à plus de deux modes possibles, de la forme :

tuple [mb, mb]

3 - En BASEL, on peut avoir un contexte correct admettant plusieurs modes dans deux cas :

1er Cas : tous les contextes où le langage définit les modes m1, m2, ..., mp, (p > 1) possibles.

2ème Cas : les appels des procédures génériques définies dans le programme, admettant plusieurs listes de modes de paramètres.

Dans le premier cas, d'après la définition de BASEL, la situation est toujours analogue à celle de IV.2,1 et de IV.2,2 ci-dessus, car :

- soit on fait un premier appel avec un des modes possibles comme paramètre, et si le résultat est faux, on vérifie simplement les autres modes par égalité (Voir Ch. 3, IV.2,1).

- soit on fait un premier appel pour vérifier une certaine construction, ensuite un deuxième appel avec un des modes possibles comme paramètre, et ensuite on vérifie les autres par égalité si nécessaire (Voir Ch. 3, IV.2,2).

On peut donc conclure :

(1) il n'est pas nécessaire de garder le mode n avant chaque appel de la procédure MODIFICATION ;

(2) il n'est pas nécessaire de supprimer les modifications générées si le résultat d'un appel est faux :

soit elles servent dans le cas d'une autre vérification ultérieure, soit le contexte, donc le programme, est incorrect, et le code généré ne peut être exécuté.

Dans le cas des appels des procédures génériques, il est évident qu'on ne peut pas appliquer le même raisonnement car il n'y a aucun rapport entre les différents modes possibles définis dans le programme. Cela implique que chaque vérification doit se faire avec le mode de départ, les modifications faites lors d'un appel ne peuvent pas être utilisées pour un autre.

IV.3 - Appels de MODIFICATION(m, n) au cours de la compilation -

IV.3.1 - Opérandes -

IV.3.1.1 - Suite de deux valeurs -

Dans la suite valeur valeur, la première valeur doit être une procédure avec paramètres ou une séquence (Voir Ch.1, VII.3.1.a).

L'appel est donc :

```
si MODIFICATION(proc [x] x , n)
alors B
sinon si MODIFICATION (seq x, n)
      alors B
      sinon E ;
```

IV.3.1.2 - Suite opérateur opérande -

Dans la suite opérateur opérande, l'opérateur détermine le mode de 1'opérande (Voir App. II) :

IV.3.1.2.1 - Opérateurs sur pointeurs -

- alloc

  On n'appelle pas MODIFICATION, car on doit seulement vérifier m ≠ none.

- set

C'est le mode de la valeur repérée par le premier composant de la tuple qui détermine le mode du deuxième composant.

Exemple :

set (proc ptr union [int, real], int)

L'algorithme de l'appel est donc :

```
si MODIFICATION (tuple [ptr χ, χ], n)
alors  soient n1, n2 les modes composants de n
       (le transforme  du paramètre original) ;
       n1 ← DP(n1) ;
       si MODIFICATION (n1, n2)
       alors B
       sinon E
sinon E ;
```

- same.

same est un opérateur de comparaison de deux valeurs de mode ptr p. L'opérande de same est double (Voir Ch.1,V.1.3)et des modifications sont permises pour chacune des valeurs qui le composent afin d'avoir deux valeurs de mode ptr p, pour p ≠ none.

Si n est le mode de l'opérande, un premier appel permet la transformation de n en tuple [ptr χ, ptr χ].

Si n1 et n2 sont les modes des éléments de n après l'appel, deux cas peuvent se présenter :

n1 = n2 : ce qui signifie que les deux valeurs sont un argument correct pour same.

n1 ≠ n2 : il est nécessaire de faire des modifications afin de décider si l'opérande est correct ou non.
Le choix entre la transformation de n1 en n2, de n2 en n1 ou de n1 et n2 en un troisième mode se fonde sur le fait suivant :
pour deux modes p1 et p2 de type m1, tels que p2 = p(p1) (partie de p1, voir Ch.2,II.3.b) p2 est représenté avant p1 dans le tableau C-L

(Voir Ch.IV,2.2) c'est-à-dire p2 < p1. L'affirmation réciproque n'est pas vrai :

p2 < p1 $\not\Rightarrow$ p2 = p(p1).

Dans le cas de same on modifie le plus grand pour essayer de le ramener au plus petit, en ayant toujours deux modes de la forme : ptr ....

Par exemple, si n1 > n2 on modifie n1, si au cours de la modification n1 < n2, on modifie n2, et ainsi de suite jusqu'à l'obtention de l'égalité ou jusqu'à la décision que l'opérande est incorrect.

L'algorithme de l'appel est :

```
(1)→ L : si MODIFICATION (tuple [ptr ℓ, ptr ℓ], n)
(2)→     alors  soient n1 et n2 les modes des composants de n
         (transformé) ;
(3)→            si n1 = n2
(4)→            alors B
(5)→            sinon si n1 < n2
(6)→                  alors (n ← tuple [n1, DR (n2)] ; aller à L)
(7)→                  sinon (n ← tuple [DR (n1), n2] ; aller à L)
(8)→     sinon E ;
```

Les exemples suivants montrent différents cas de modifications :

1 - n = tuple [ptr bool, ptr bool]

exécution des pas (1), (2), (3) et (4) : n est correct.

2 - n = tuple [ptr proc bool, ptr ptr proc bool]

exécution des pas (1), (2), (3), (5), (6), (1), (2), (3), (4) : n est correct

3 - n = tuple [ptr proc ptr int, proc ptr ptr int]

On détaille l'exécution de l'algorithme d'appel :

(1)→ n = tuple [ptr proc ptr int, ptr ptr int]

(2)→ n1 = ptr proc ptr int et n2 = ptr ptr int

(3)→

(5)→ deux alternatives peuvent se présenter :

n1 < n2

(6)→ n = tuple [ptr proc ptr int, ptr int]

(1)→ n = tuple [ptr proc ptr int, ptr int]

(2)→ n1 = ptr proc ptr int et n2 = ptr int

(3)→

(5) forcément : n1 > n2

(7)→ n = tuple [proc ptr int, ptr int]

(1)→ n = tuple [ptr int, ptr int]

(2)→ n1 = ptr int et n2 = ptr int

(3)→

(4)→ n1 = n2

ou

n1 > n2

(7)→ n = tuple [proc ptr int, ptr ptr int]

(1)→ n = tuple [ptr int, ptr ptr int]

(2)→ n1 = ptr int et n2 = ptr ptr int

(3)→

(5)→ forcément n1 < n2

(6)→ n = tuple [ptr int, ptr int]

(1)→ n = tuple [ptr int, ptr int]

(2)→ n1 = ptr int et n2 = ptr int

(3)→

(4)→ n1 = n2

4 - n = tuple [proc ptr ptr proc ptr bool, ptr proc proc ptr real]

n n'est pas correct,

on sortira par (8) après avoir transformé n en

tuple [ptr bool, real] ou en

tuple [bool, ptr real].

IV.3.1.2.2 - Opérateurs sur des séquences -

- length.

L'opérande doit être une séquence, on ne s'intéresse pas au mode de ses éléments. L'appel est :

```
si MODIFICATION (seq χ, n) alors B sinon E ;
```

- change.

Le premier composant de l'opérande doit être une séquence, on ne s'intéresse pas au mode de ses éléments. L'appel est :

```
si MODIFICATION (tuple [seq χ , int], n) alors B sinon E ;
```

IV.3.1.2.3 - Opérateurs déterminant un seul mode -

m est défini d'après la liste donnée dans l'appendice II. Par exemple, dans le cas de and, l'appel est :

```
si MODIFICATION (tuple [bool, bool], n) alors B sinon E ;
```

et pour float :

```
si MODIFICATION (int, n) alors B sinon E ;
```

IV.3.1.2.4 - Opérateurs déterminant plusieurs modes -

- plus, mins, mul, div :

```
si MODIFICATION (tuple [χ, χ], n)
alors si MODIFICATION (tuple [int, int],n)
      alors B
      sinon si n = tuple [real, real]
            alors B
            sinon E
sinon E ;
```

- exp

```
si MODIFICATION (tuple [χ, χ ], n)
alors si MODIFICATION (tuple [real, int], n)
      alors B
      sinon si n = tuple [int, int]
            alors B
            sinon E
sinon E ;
```

- abs, sign

```
si MODIFICATION (int, n)
alors B
sinon si n = real
      alors B
      sinon E ;
```

- eq, neq

```
si MODIFICATION (tuple [χ, χ], n)
alors si MODIFICATION (tuple [int, int], n)
      alors B
      sinon si n = tuple [real, real]
            alors B
            sinon si n = tuple [bool, bool]
                  alors B
                  sinon si n = tuple [char, char]
                        alors B
                        sinon E
sinon E ;
```

- less, leq, grt, geq

même appel que pour eq sans la possibilité tuple [bool, bool]

- cread

```
si MODIFICATION (int, n)
alors B
sinon si MODIFICATION (tuple [int, int], n)
      alors B
      sinon E ;
```

- write

```
si MODIFICATION (tuple [int, ℵ], n)
alors si MODIFICATION (tuple [int, int], n)
      alors B
      sinon si n = tuple [int, real]
            alors B
            sinon si n = tuple [int, bool]
                  alors B
                  sinon si n = tuple [int, char]
                        alors B
                        sinon si MODIFICATION
                       (tuple [int, tuple [char, ..., char]],n)
                              alors B
                              sinon E
sinon E ;
```

IV.3.2 - Boucles -

La valeur qui précéde times doit être un entier :

```
si MODIFICATION (int, n) alors B sinon E ;
```

IV.3.3 - Choix d'un champ d'une valeur structurée -

S'il s'agit d'un nom, c'est-à-dire de la partie gauche d'une affectation, on ne permet pas de modifications. Après of on doit avoir un identificateur de mode struct [...]. Il n'est pas nécessaire d'appeler la procédure pour faire la vérification.

Dans le cas général des expressions, la valeur qui suit of doit être de mode ptr struct [...] ou de mode struct [...].

L'appel est donc :

```
L : si MODIFICATION (ptr χ, n)
    alors ( n1 ← DR(n)
            si n1 = struct [...]
            alors B
            sinon (n ← n1 ; aller à L))
    sinon si n = struct [...]
          alors B
          sinon E ;
```

IV.3.4 - Affectation -

```
m ← mode de la partie gauche ;
n ← mode de la partie droite ;

si MODIFICATION (m, n) alors B sinon E ;
```

IV.3.5 - Appel de procédure avec paramètres -

a) - procédure non générique -

```
m ← mode de la liste de paramètres de la procédure ;
n ← mode courant ;

si MODIFICATION (m, n) alors B sinon E ;
```

b) - procédure générique -

soient : m1, m2, ..., mp les modes pour les différentes listes possibles des paramètres, l'appel est :

```
    i ← 1 ;
    ng ← n ;
L : si MODIFICATION (mi, n)
    alors (pour j ← i + 1 jusqu'à p faire :
                (n ← ng ;
                 si MODIFICATION (mj, n) alors E);
          B ;
    i ← i + 1 ;
    si i ≤ p alors (n ← ng ; aller à L) ;
    E ;
```

Dans ce cas, qui est le seul où la recherche des modifications n'est pas déterministe, il faudrait que cette séquence supprime également les instructions générées dans le cas d'un appel infructueux de MODIFICATION.

IV.3.6 - Choix des éléments d'une séquence -

La deuxième valeur, dans la suite valeur valeur doit être de mode int ou de mode tuple [int, int], si la première est de mode seq....

```
si MODIFICATION (int, n)
alors B
sinon si MODIFICATION (tuple [int, int], n)
      alors B
      sinon E ;
```

IV.3.7 - Symbole --> -

La valeur qui précéde le symbole --> doit être booléenne.

```
si MODIFICATION (bool, n) alors B sinon E ;
```

IV.3.8 - Symbole ; (suppression) -

```
L : si MODIFICATION (proc ℵ , n)
    alors  n   AP(n) ;
           aller à L
    sinon n ← SP (n) ;
```

Une valeur est supprimée quand son mode est compatible avec none, c'est-à-dire quand il n'est pas possible de la transformer en une procédure sans paramètres. Si elle est compatible avec une procédure sans paramètres, la procédure doit être appelée, et c'est sur son résultat que l'opération de suppression doit s'appliquer.

IV.3.9 - Modification explicite -

D'après la syntaxe, une expression de mode n peut être précédée par un mode m afin de spécifier la transformation de v_n en v_m. L'appel est :

```
si MODIFICATION (m, n) alors B sinon E ;
```

CHAPITRE 4

ARCHITECTURE GENERALE DU COMPILATEUR

Les algorithmes présentés aux chapitres 2 et 3 ont été utilisés pour écrire un compilateur de BASEL. Il s'agit d'un compilateur en un seul passage qui traduit le langage BASEL en un langage de pseudo-code, qui doit être interprété.

Dans ce qui suit on s'intéresse à la description des composants fondamentaux du compilateur : les tableaux qu'il utilise, le traitement des déclarations, la détermination et la vérification des modes, la génération du pseudo-code, etc....

I - NOTATION -

Tout au long de ce chapitre on adopte les conventions suivantes :

n représente un entier.

a représente une chaîne des caractères.

b représente une valeur booléenne : "vrai", notée par 1 ou "faux", notée par 0.

α_P représente une adresse relative de la pile d'exécution (voir Ch.1, I.1.4).

α_L représente une adresse de la mémoire libre à l'exécution (voir Ch.1, I.1.4).

α_C représente une adresse de la mémoire où est rangé le pseudo-code.

II - REPRESENTATION DES VALEURS -

Soit u une unité de mémoire dans laquelle on peut ranger un caractère (p.e. u = octet).

Pour la taille de valeurs on adopte les conventions suivantes.

Notation : la taille de la mémoire nécessaire pour ranger $v_{\underline{m}}$ est notée : $ta(v_{\underline{m}})$.

$ta(v_{\underline{int}}) = 4$ u ;

$ta(v_{\underline{real}}) = 4$ u ;

$ta(v_{\underline{bool}}) = 1$ u ;

$ta(v_{\underline{char}}) = 1$ u ;

$ta(v_{\underline{ptr}\ \underline{m}}) = 4$ u ;

$ta(v_{\underline{seq}\ \underline{m}}) = 12$ u dont : 4 u pour indiquer l'adresse de la mémoire libre où se trouve le premier élément de la séquence (α_L),
4 u pour indiquer sa longueur,
4 u pour indiquer $ta(v_{\underline{m}})$;

$ta(v_{\underline{tuple}\ [\underline{m1}, \ldots, \underline{mn}]}) = \sum_{i=1}^{n} ta(v_{\underline{mi}})$;

$ta(v_{\underline{struct}\ [\underline{m1}\ s1, \ldots, \underline{mn}\ sn]}) = \sum_{i=1}^{n} ta(v_{\underline{mi}})$;

$ta(v_{\underline{union}\ [\underline{m1}, \ldots, \underline{mn}]}) = \max_i ta(v_{\underline{mi}}) + 1$ u, 1 u sert à ranger l'indication du mode $\underline{mi}$ de la valeur courante ;

$ta(v_{\underline{proc}\ [\underline{m1}, \ldots, \underline{mn}]\ \underline{M}})$ ou $ta(v_{\underline{proc}\ \underline{M}})$ = 8 u dont : 4 u pour indiquer l'adresse où se trouve le corps de la procédure (α_C),
4 u pour indiquer l'adresse où se trouve la liste des pointeurs qui repèrent les valeurs des variables globales (α_L),

III - TABLEAUX -

On décrit la structure des principaux tableaux utilisés par le compilateur.

III.1 - Représentation et traitement des modes -

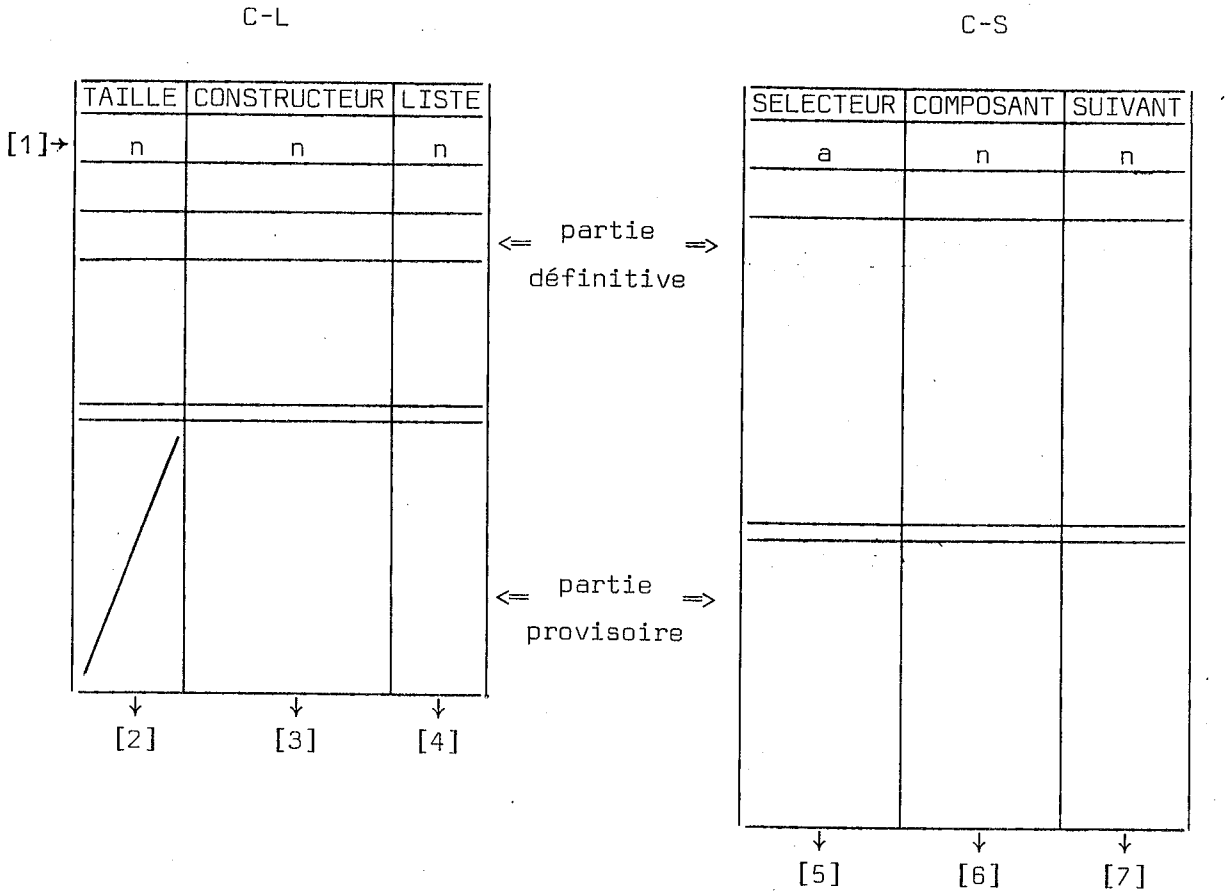
III.1.1 - Représentation et étude de l'équivalence : C-L, C-S et M-U.

La représentation des modes se fait conformément au chapitre 2 : les représentations de type ml sont rangées sous forme de listes dans les tableaux C-L et C-S. Celles de type mu sont rangées dans la matrice booléenne M-U.

Les tableaux C-L, C-S et M-U ont chacun deux parties : une partie provisoire et une partie définitive. Les représentations des modes entièrement connus au moment de leur traitement sont rangées dans la partie définitive correspondant à leur type (ml ou mu), tandis que les représentations des modes qui font référence à des modes encore inconnus au moment de leur traitement sont rangées dans la partie provisoire. On dira qu'un mode m est définitif (ou provisoire) si sa représentation est rangée dans les parties définitives (ou provisoires).

En plus de la structure décrite dans le chapitre 2, pour toute représentation du mode m, on garde aussi l'indication de la taille d'une valeur du mode m.

La structure des tableaux est donc la suivante :



[1] - Chaque entrée de C-L correspond à la représentation d'un mode m de type ml.

[2] - $ta(v_m)$.

[3] - Code du constructeur c ; c ∈ {ptr, seq, tuple, struct, proc}

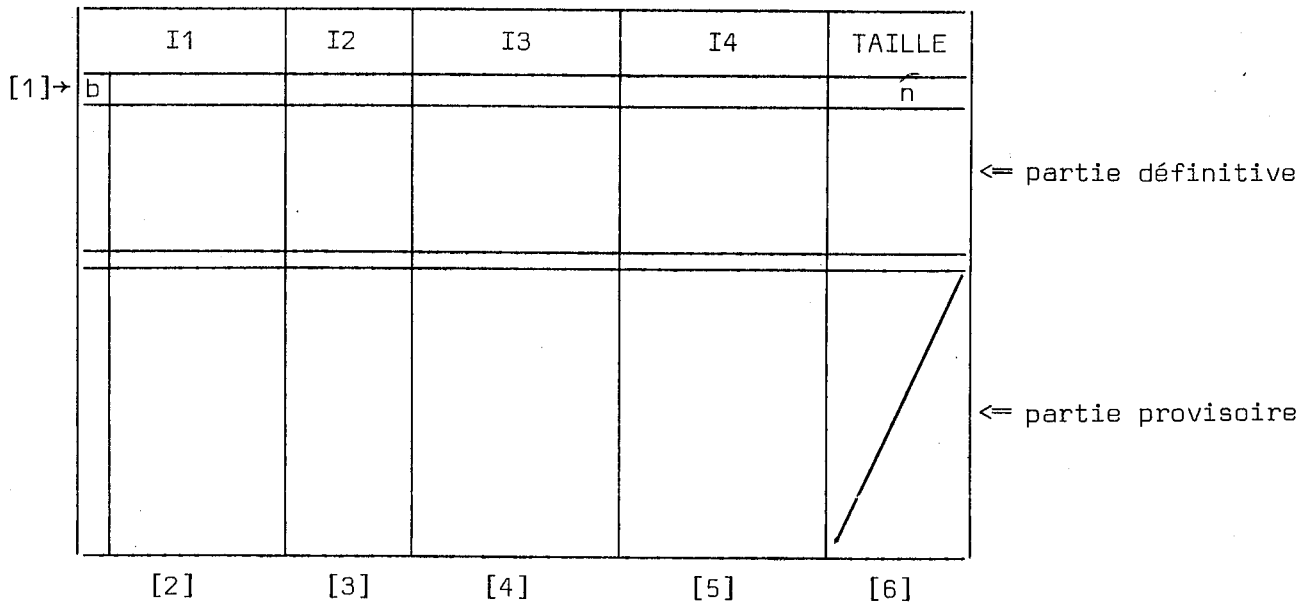
[4] - Pointeur vers C-S.

[5] - Identificateur du sélecteur de champ, dans le cas c = struct.

[6] - Soit pointeur vers C-L (ml), soit pointeur vers M-U (mu), soit pointeur vers M-D (md), soit mb.

[7] - Soit pointeur vers C-S, soit 0 qui indique la fin de la liste.

M-U



[1] - Chaque ligne de M-U correspond à la représentation d'un mode m de type mu.
Si comp(j) de m = 1 le mode indiqué par j est un des modes composants de l'union m.

[2] - si $j \in I1$, j est un pointeur vers M-D (md)

[3] - si $j \in I2$, j est un mb

[4] - si $j \in I3$, j est un pointeur vers C-L (ml)

[5] - si $j \in I4$, j est un pointeur vers M-U (mu)

[6] - $ta(v_m)$

Remarque : M-U(m, j) = 0 pour $j \in I4$ si on a fait le remplacement d'unions (voir Ch. 2, I.2.2).

III.1.2 - Vérification des conditions de définition des modes : R -

On utilise la matrice booléenne R pour indiquer la dépendence des modes déclarés.

Les lignes R correspondent aux symboles m déclarés, les colonnes aux symboles n utilisés dans la partie mode de la déclaration de

m. R(m, n) = 1 si m = φ (n), et il n'existe pas de constructeur ptr dans le chemin qui lie (m) à n (voir Ch.1, II.3.4.4). Autrement R(m, n) = 0.

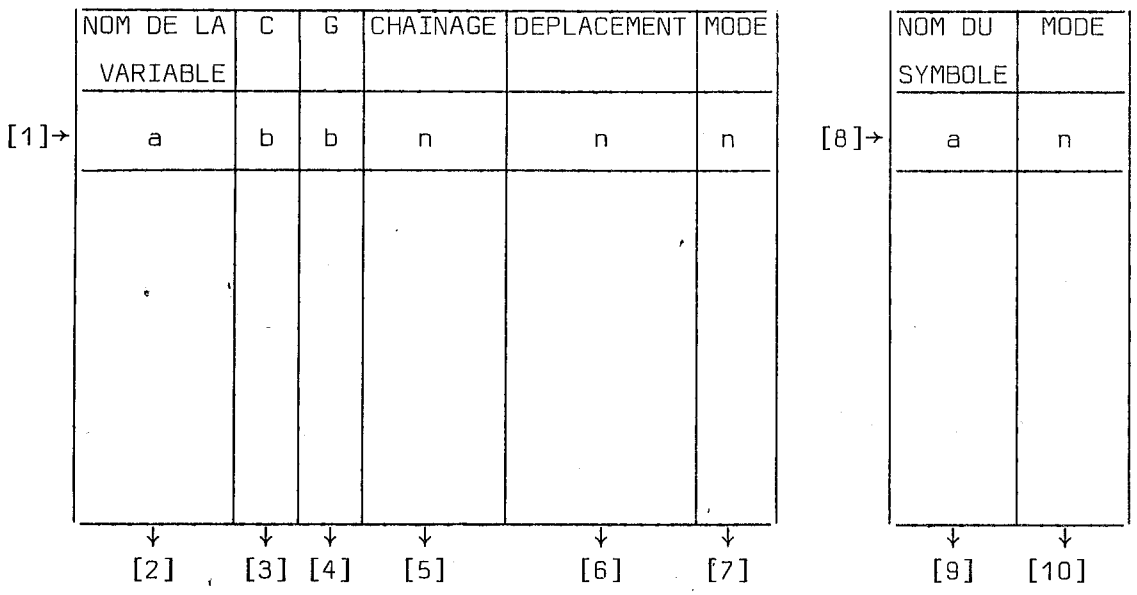
III.2 - Traitement des déclarations : V-D et M-D -

Les variables et les symboles déclarés sont rangés, accompagnés de l'indication de leur mode, dans les tableaux V-D et M-D.

Le rangement dans V-D et dans M-D se fait d'après la structure des blocs du programme BASEL compilé, ce qui implique des renseignements sur le niveau des blocs. A chaque moment de la compilation V-D et M-D ont toutes les informations sur les variables et symboles déclarés du bloc en cours de traitement et des blocs englobants.

En plus pour les variables génériques et pour les variables déclarées au moyen de la déclaration conditionnelle il existe des renseignements **supplémentaires**.

La structure de V-D et de M-D est la suivante :



[1] - Chaque entrée de V-D correspond à une variable ou à l'indication du début de bloc imbriqué.

[2] - Soit un identificateur de variable, soit "....." qui indique le début d'un bloc imbriqué.

[3] - Indicateur de variable déclarée au moyen de la déclaration conditionnelle.

[4] - Indicateur de variable générique.

[5] - Chaînage des différentes significations d'une même variable (pour les génériques).

[6] - Déplacement qui indique l'emplacement dans la pile, relatif au début du bloc, où la valeur de la variable sera rangée au moment de l'exécution.

[7] - Mode : mb, ml (pointeur vers C-L), mu (pointeur vers M-U) ou md (pointeur vers M-D).

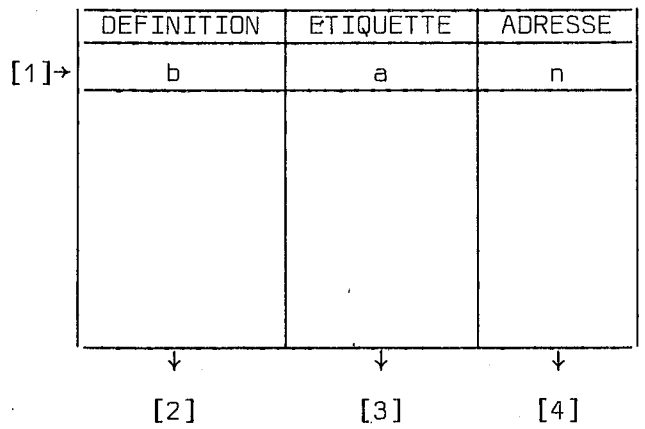
[8] - Chaque entrée de M-D correspond à un symbole ou à l'indication du bloc imbriqué.

[9] - Soit un symbole, soit "....." qui indique le début d'un bloc imbriqué.

[10]- Mode : mb, ml (pointeur vers C-L), mu (pointeur vers M-U) ou md (pointeur vers M-D).

III.3 - Traitement des étiquettes : ET -

Les renseignements concernant les étiquettes sont rangés dans ET :



[1] - Chaque entrée de ET correspond à un identificateur d'étiquette ou à l'indication d'un début de bloc.

[2] - 1 indique que l'étiquette est déjà définie,
0 indique que l'étiquette n'est pas encore définie, mais qu'elle a été utilisée.

[3] - Identificateur d'étiquette, ou

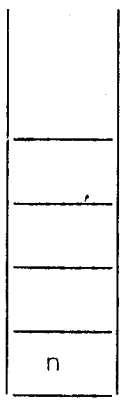
symbole |, considéré comme un cas particulier d'une étiquette (référencée par →), ou

indication du début de bloc : ".....".

[4] - Si la valeur de [2] est 1, [4] représente l'adresse α_c repérée par l'étiquette de [3] ;
Si la valeur de [2] est 0, [4] représente l'adresse α_c où se trouve la première instruction rencontrée qui fait référence à l'étiquette. Cette instruction est chaînée avec toutes les autres qui utilisent la même étiquette, tant que la définition n'est pas rencontrée.

III.4 - La pile -

Le compilateur utilise une pile, qui est en quelque sorte l'image de la pile au moment de l'exécution. Alors qu'à l'exécution il y aura au sommet de la pile la dernière valeur élaborée, à la compilation il y a le mode de la dernière expression compilée.



PILE

III.5 - Tableaux des littéraux -

Les littéraux rencontrés au cours de la compilation sont rangés dans des tableaux, d'après leur mode. Ils sont identifiés par leur position dans les tableaux. Ainsi, il existe les tableaux TAB_INT, TAB_REAL et TAB_CHAINE où on garde respectivement des entiers, des réels et des chaînes des caractères.

IV - LA COMPILATION -

On distingue deux phases : l'analyse syntaxique et le traitement sémantique.

IV.1 - Analyse syntaxique -

Elle est faite par un analyseur déterministe, d'après une grammaire LL(1) équivalente à celle de BASEL (voir App. I).

IV.2 - Traitement sémantique -

Il se fait au moyen de routines qui sont appelées au cours de l'analyse syntaxique.

Dans ce qui suit on détaille les aspects les plus importants du traitement sémantique.

IV.2.1 - Traitement des modes -

Il comporte : rangement des représentations dans les tableaux, étude de l'équivalence, transfert de représentations, réduction, etc....

Les procédures suivantes servent à décrire le traitement des modes :

REPRESENTATION : procédure qui range la représentation d'un mode dans les parties provisoires des tableaux, au fur et à mesure que le mode se définit.

REMPLACEMENT (m) : m est de type mu, c'est-à-dire m = union [m1, m2, ..., mn] Pour tout i $\in$ [1, n] tel que mi = union [mi1, mi2, ..., mini] on fait le remplacement (voir Ch.2,I.2.2).

EQUIVALENCE (m, n) : si m $\doteq$ n (voir Ch.2, II.1) la procédure donne le résultat vrai, sinon elle donne le résultat faux. Cette procédure est faite d'après l'algorithme présenté dans Ch.2, IV.2.

PASSAGE (m) : procédure qui transfère la représentation de m, rangée dans les parties provisoires aux parties définitives. Si m = φ(n) et n est

rangé dans les parties provisoires, PASSAGE (m) entraîne PASSAGE (n). Si m1 = p(m) et m1 est rangé dans les parties définitives PASSAGE (m) n'entraîne pas le transfert de m1.

REFERENCE (m, n) : lorsqu'on traite une déclaration de mode, où m est le symbole déclaré et m = φ (n) tel qu'il n'existe pas de ptr dans le chemin qui lie (m) à n (voir Ch.1, II.3.4.4) on définit R(m, n) = 1 ; autrement R(m, n) = 0.

RESTRICTION : elle calcule la fermeture transitive de R : R*. R* indique toutes les références directes ou indirectes de m à n pour tout m déclaré, tel qu'il n'existe aucun ptr entre (m) et n.

Remarque : si R*(m, m) = 1 pour un certain m, m est incorrect.

REDUCTION (m) : cherche p(m), tel que p(m) $\neq$ m et p(m) est minimale (voir Ch.2, V.3).

IV.2.2 - Traitement des déclarations -

Le traitement des déclarations se fait en deux étapes :

1ère étape : au fur et à mesure de l'analyse des déclarations,

2ème étape : à la fin de la partie déclarations du bloc.

IV.2.2.1 - Première étape -

Elle consiste en la réalisation des pas suivants pour chaque déclaration :

E1.1 rangement de l'identificateur de variable ou du symbole dans V-D ou dans M-D.

E1.2 indication des variables génériques ou conditionnelles.

E1.3 rangement du mode, au moyen de REPRESENTATION s'il est construit.

Toute référence à un symbole n est rangée comme une référence de type mb, ml ou mu si n est entièrement connu, ou comme une référence md si n n'est pas encore connu ou est en cours de définition ou est provisoire.

Chaque fois qu'une représentation d'un mode m est entièrement rangée :

E1.3.1 s'il n'y a pas des références md.

E1.3.1.1 REMPLACEMENT (m) si nécessaire.

E1.3.1.2 si EQUIVALENCE (m, n) pour un certain n définitif, on remplace la référence à m par une référence à n. Si non, PASSAGE (m) et on remplace la référence à m. On libère la place de m dans les parties provisoires.

E1.3.1.3 on continue le rangement du mode, à partir de E1.3, si la déclaration n'est pas finie ou on traite une nouvelle déclaration, à partir de E1.1, si elle est finie.

E1.3.2 s'il y a des références md : la représentation du mode reste dans les parties provisoires, et on continue soit à partir de E1.3, si la déclaration n'est pas finie ou de E1.1 pour traiter une nouvelle déclaration, si elle est finie, comme dans E1.3.1.3.

IV.2.2.2 - Deuxième étape -

Elle est faite à la fin de la partie déclarations d'un bloc : à ce moment on a tous les renseignements sur les symboles déclarés et on peut aborder le traitement des modes qui ont une représentation provisoire.

Elle consiste en :

E2.1 - RESTRICTION.
Si $R^*(m, m)$ pour un certain m, m est incorrect.

E2.2 - S'il existe m référencé par un certain mode, tel que m n'est pas déclaré ni dans le bloc actuel ni dans les englobants, m est incorrect.

E2.3 - Pour tout m (correct, d'après E2.1 et E2.2 si m est un symbole), tel que m est mu et la représentation de m est provisoire : REMPLACEMENT (m).
Cela est toujours possible ; d'après la vérification de RESTRICTION, on ne peut pas boucler indéfiniment.

E2.4 - Transfert des représentations provisoires :

E2.4.1 - Pour tout symbole m dont la représentation est provisoire : REDUCTION (m) et
PASSAGE (m), s'il n'existe aucun n tel que
EQUIVALENCE (m, n) et n est définitif.
Il est à noter que le transfert de la représentation de m peut entraîner le transfert des représentations de n1, de n2, ..., de np si m = φ (ni), i $\in$ [1, p]. PASSAGE (m) appelle REDUCTION (ni), i $\in$ [1, p] avant de faire le transfert.

E2.4.2 - Pour tout mode m des variables déclarées dont la représentation est provisoire, on fait PASSAGE (m) s'il n'existe aucun n tel que EQUIVALENCE (m, n) et n est définitif.

E2.5 - Calcul des déplacements des variables déclarées, chaînage des variables génériques, génération du pseudo-code concernant la vérification déterminée par les déclarations conditionnelles.

IV.2.2.3 - Remarques sur le traitement des déclarations -

1 - Les parties provisoires des tableaux C-L, C-S et M-U sont libérées à la fin de la partie déclarations de chaque bloc.
Le contenu des parties définitives est utilisé tout au long de la compilation d'un programme.

2 - Toutes les références a un symbole n inconnu ou provisoire sont de la forme md, c'est-à-dire un pointeur vers M-U. Dès que la représentation de n devient définitive c'est l'indication de mode qui accompagne n qui est changée et non toutes les références à n. Les références à n dans les modes m = φ (n) seront traitées après la définition définitive de n.

3 - L'appel de REDUCTION (m) pour tout symbole m déclaré est en autre une solution au problème de l'équivalence des modes récursifs.
Si m1, m2, ..., mn sont récursifs : mi = φ (mj), j $\in$ [1, n] pour tout i $\in$ [1, n]
et mi $\neq$ mj pour tout i $\in$ [1, n] et tout j $\in$ [1, n] l'algorithme de réduction trouve la représentation minimale pour mi, i $\in$ [1, n]. Or, contrairement à [MA] il n'est pas nécessaire de distinguer le traitement des modes récursifs.

4 - Il est à noter qu'il n'est pas nécessaire d'appeler REDUCTION (m) si m est le mode de la déclaration d'une variable, car la possibilité de réduction ne peut s'appliquer qu'aux symboles déclarés.

5 - Le traitement des modes implique la conservation de certaines "traces", comme on l'a déjà signalé au chapitre 2. L'inspection des traces sert à éviter des comparaisons répétées (voir Ch.2,IV.3).

6 - Il se peut que pour un certain m rangé dans une partie provisoire, il existe p(m) rangée dans une partie définitive. PASSAGE (m) fait seulement le transfert des représentations provisoires.

7 - De E1.3.1.2 et de E2.4 on déduit que la représentation de p(m) précéde à celle de m pour tout m.

IV.2.3 - Détermination des modes -

Pendant la compilation des expressions, leur mode est construit dans la partie provisoire des tableaux. Chaque fois que la représentation d'un mode m est terminée, on fait PASSAGE (m) s'il n'existe aucun n définitif tel que EQUIVALENCE (m, n).

Le traitement du mode des expressions est beaucoup plus simple que celui du mode des déclarations, du fait que la partie_déclarations précède la partie_instructions. Ainsi, elle donne tous les renseignements nécessaires pour la compilation des expressions : le mode des variables et des symboles est connu à ce moment. C'est une des différences les plus importantes avec Algol 68. Elle permet la compilation en un seul passage.

Dans le traitement des modes : la comparaison d'unions se réduit à la comparaison de deux vecteurs booléens ; le problème de REDUCTION ne se pose pas, car il se présente dans les déclarations de modes m = φ (m).

Quelques symboles du langage agissent particulièrement sur la construction des modes :

( ) [ ] sur la construction de tuple [...], et d'union [...],

, sur la définition des modes de la tuple,

| sur la définition des modes de l'union,

< > sur la définition de proc...,

IV.2.4 - Vérification des modes -

Dans le chapitre 3 (Ch.3, III et Ch.3, IV) on a décrit la vérification des modes au cours de la compilation, à l'aide de la procédure MODIFICATION (m, n).

Du fait que :

- il n'existe qu'une représentation pour tous les modes équivalents,
- il n'existe pas des représentations redondantes,
- le remplacement d'unions est fait,

la comparaison des modes imposée par MODIFICATION (m, n) est très simple. Dans la plupart des cas, elle se réduit à la vérification de l'égalité de deux entiers, qui identifient les représentations des modes.

De l'étude de la forme de t tel que t(n) = m pour certains $m \in \mathcal{M}$ et de l'analyse de certains contextes particuliers (voir Ch.3, IV.1 et Ch.3, IV.2) on déduit que la modification en BASEL est déterministe dans tous les cas définis dans le langage, y compris le cas des contextes admettant plusieurs modes. C'est-à-dire que pour transformer v_n en v_m il y a au plus un chemin possible. Si on n'arrive pas à faire la transformation le programme est incorrect. La modification imposée par les appels des procédures génériques est un cas particulier des contextes admettant plusieurs modes ; on ne peut pas assurer le déterminisme car la modification dépend du programme à compiler. On la considère non déterministe et c'est le seul cas ou un "retour arrière" dans la génération s'impose (voir Ch.3, IV.3.5).

IV.2.5 - Génération du pseudo-code -

Le pseudo-code est de la forme :

O
ou O d
ou O d1 d2

où O symbolise une opération et d, d1, d2 sont des opérandes. L'opération implique parfois la considération d'autres opérandes : ce sont des valeurs qui se trouvent au sommet de la pile d'exécution.

La description du pseudo-code est donnée dans l'appendice 4, selon les expressions BASEL qu'ils traduisent. Dans l'appendice 5, on présente un exemple de génération (sortie du compilateur).

V - L'INTERPRETEUR -

L'interpréteur du pseudo-code est très simple, comme on peut le remarquer d'après sa signification (voir App. 4). La seule difficulté de l'interprétation est l'écriture d'un algorithme de récupération de mémoire libre.

CONCLUSION

Sur le travail qui vient d'être présenté, il peut être intéressant de faire les quelques remarques suivantes :

- La formalisation qui a été faite pour le traitement des modes a permis d'aborder de façon systèmatique les composants fondamentaux du compilateur. En particulier, l'algorithme de réduction des représentations apporte une solution simple au problème du traitement des modes récursifs. Les divers résultats obtenus à ce sujet peuvent être utilisés également dans la réalisation d'un compilateur pour Algol 68.

- La façon dont a été étudiée le problème des modifications a mis en valeur la cohérence du langage et a permis de démontrer que le choix des modifications à appliquer peut toujours être fait de façon déterministe. Les algorithmes qui ont été proposés sont des algorithmes généraux, donc peu efficaces dans certains cas. Mais, comme on l'a montré, il est aisé de traiter ces divers cas au moyen d'algorithmes particuliers. La formalisation qui a été faite correspond aux idées présentées par Scheidig [SC], et il apparaît que les modifications en BASEL sont beaucoup plus simples à traiter qu'en Algol 68, bien qu'elles remplissent un rôle équivalent.

- En décrivant l'architecture du compilateur, on a pu montrer comment utiliser pratiquement les idées exposées par ailleurs. Actuellement, la réalisation du compilateur n'est pas achevée, les problèmes restant à résoudre étant d'importance mineure. Ce compilateur est de nature expérimentale, et, en général, la recherche de l'efficacité n'a pas été prise en considération, si bien qu'une implémentation définitive et pratiquement utilisable de BASEL en demanderait une réécriture à peu près complète, où l'on utiliserait bien sur les mêmes algorithmes généraux. Quant à l'interpréteur, sa structure est très simple et il ne pose aucun problème nouveau. La seule difficulté liée à l'organisation au cours de l'exécution tient à la nécessité d'un algorithme de gestion de la mémoire.

- Enfin en ce qui concerne BASEL, il apparaît que c'est un langage qui avec une structure remarquablement simple, introduit de façon rigoureuse un ensemble très riche de concepts de programmation évoluée. D'un point de vue pédagogique, son intérêt semble être au moins aussi grand que celui d'Algol 68.

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

[CU] - CUNIN Pierre-Yves
Communication interne - Université Scientifique et Médicale de Grenoble
Février 1972.

[JH] - JORRAND Philippe and HAMMER Michaël
"A BASEL Primer", Massachussets Computer Associates Report, 1969.

[JO] - JORRAND Philippe
"Some Aspects of BASEL, the base language for an extensible language facility".
Extensible languages Symposium, BOSTON.
August 1969, SIGPLAN Notices, Vol. 4, N° 8, pp.14-17.

[KN] - KNUTH Donald
"The Art of Computer Programming".
Vol. 1, Section 2.3.5 Exercise 11.
Addison-Wesley, Second Printing 1969.

[KO] - KOSTER C. H. A.
"On infinite modes"
Algol Bulletin, 30, pp. 86-89.
February 1969, Amsterdam

[MA] - MARCHE Gérard
"Organisation de la table des déclareurs de mode. Equivalence de mode en Algol 68".
Rapport de D. E. A. - Université Scientifique et Médicale de Grenoble.
Juin 1972.

[SC] - SCHEIDIG H

"Syntax and mode check in an Algol 68 compiler".

Proceedingsof the IFIP Working Conference on Algol 68 Implementation

Munich, July 20-24, 1970, pp. 83-92.

[WI] - Van WHNGAARDEN A, MAILLOUX B. J., PECK J. E. L. and KOSTER C. H. A.

"Report on the Algorithmic Language ALGOL 68"

Mathematisch Centrum, Amsterdam.

October 1969, 2nd Edition, MR 101.

APPENDICE I

S Y N T A X E D E B A S E L

programme → expression_composée

expression_composée → ({bloc}*)

bloc → {partie_déclarations in | ∈} partie_instructions

partie_déclarations → déclaration {, déclaration}*

partie_instructions → branche { | branche}*

branche → suite_d'_instructions {—> suite_d'_instructions}*

suite_d'_instructions → instruction {; instruction}*

instruction → {étiquette}* clause

étiquette → identificateur :

déclaration → déclaration_de_variable |
déclaration_de_mode

déclaration_de_variable → variable is mode |
variable mustbe mode |
variable means mode |
variable is generic

déclaration_de_mode → symbole rep mode

mode → int |
real
bool |
char |
ptr mode |
seq mode |
tuple [liste_de_modes] |
struct [liste_de_champs] |
union [liste_de_modes] |
proc [liste_de_modes] mode |
proc [liste_de_modes] none |
proc mode |
proc none |
symbole

symbole → identificateur

liste_de_modes → mode {, mode}*

liste_de_champs → champ {, champ}*

champ → mode identificateur

clause → goto identificateur |
opération_sur_fichiers |
expression

opération_sur_fichiers → {input | output | close} opérande

expression → {nom =}* term

nom → {sélecteur of}* identificateur

term → {sélecteur of}* facteur

selecteur → identificateur

facteur → {opération times}* opération

opération → {opérateur | ∈ } opérande

opérateur → alloc | set | same |
change | length |
or | and | not |
eq | neq | less | leq | grt | geq |
plus | minus | mul | div | idiv | rem | exp |
odd | abs | sign |
round | ent | float |
iread | rread | bread | cread | write

opérande → valeur {valeur}*

valeur → {mode}* unité

unité → littéral |
variable |
expression_composée |
tuple |
procédure

variable → identificateur

identificateur → lettre {lettre | chiffre | _ lettre | _ chiffre}*

procédure → < {partie_spécification in | ∈ } partie_instructions>

partie_spécification → variable is mode {, variable is mode}*

tuple → [bloc {, bloc}*]

littéral → entier |
réel |
booléen |
chaîne |
nil mode

entier → chiffre {chiffre}*

chiffre → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

réel → entier • entier

booleen → true | false

chaîne → "caracter {caracter}*"

caracter → lettre |
chiffre |
blanc |
guillemet |
autres

lettre → A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z

blanc → ␣

guillemet → ""

autres → + | − | / | * | (|) | [|] | | | : | < | = | > | ≠ |
, | . | ; | _ | '

APPENDICE II

O P E R A T E U R S

Description des opérateurs qui fournissent un résultat.
Description des opérateurs de manipulation de fichiers.

(L'explication de l'opération effectuée figure seulement dans le cas des opérateurs non usuels ou ayant une signification particulière).

OPERATIONS QUI DONNENT UN RESULTAT

Type	Opérateur	Catégorie de l'opérande	Mode de l'opérande	Mode du résultat	Description de l'opération et du résultat
SUR POINTEURS	alloc	v1	m (*)	ptr m (*)	- recherche d'un emplacement libre de taille suffisante pour une valeur de mode m, dans la mémoire libre. - rangement de v1 dans cet emplacement. - le résultat est l'adresse de l'emplacement.
	set	(v1, v2)	tuple [ptr m, m] (*)	ptr m (*)	- rangement de v2 à l'adresse v1 dans la mémoire libre. - le résultat est v1.
	same	(v1, v2)	tuple [ptr m, ptr m] (*)	bool	true si v1 et v2 sont des adresses identiques, sinon : false.
SUR SEQUENCES	length	v1	seq m (*)	int	la longueur de la séquence v1.
	change	(v1, v2)	tuple [seq m, int] (*)	seq m (*)	- la longueur de la séquence v1 est modifiée en lui ajoutant v2. - si v2 est positif, v2 élément de valeur nil m sont rajoutés à la fin de v1. Si v2 est négatif v2 éléments de la fin de v1 sont supprimés. Si v2 est nul : pas d'opération.

(*) m ≠ none

Type	Opérateur	Catégorie de l'opérande	Mode de l'opérande	Mode du résultat	Description de l'opération et du résultat
ARITHMETIQUES	plus	(v1, v2)	tuple [int, int] tuple [real, real]	int real	
	minus	(v1, v2)	tuple [int, int] tuple [real, real]	int real	
	mul	(v1, v2)	tuple [int, int] tuple [real, real]	int real	
	div	(v1, v2)	tuple [int, int] tuple [real, real]	real real	
	idiv	(v1, v2)	tuple [int, int]	int	
	rem	(v1, v2)	tuple [int, int]	int	
	exp	(v1, v2)	tuple [int, int] tuple [real, int]	real real	$v1^{v2}$
	abs	v1	int real	int real	
	sign	v1	int real	int real	-1 si v1 est négatif 0 si v1 est nul 1 si v1 est positif
	odd	v1	int	bool	true si v1 est impair false si v1 est pair

Type	Opérateur	Catégorie de l'opérande	Mode de l'opérande	Mode du résultat	Description de l'opération et du résultat
DE CONVERSION	round	v1	real	int	l'entier le plus proche de v1
	ent	v1	real char	int int	la partie entière de v1 un entier est associé de façon unique à chaque caractère possible
	float	v1	int	real	
BOOLEENS	or	(v1, v2)	tuple [bool, bool]	bool	
	and	(v1, v2)	tuple [bool, bool]	bool	
	not	v1	bool	bool	

Type	Opérateur	Catégorie de	Mode de l'opérande	Mode du résultat	Description de l'opération et du résultat
DE COMPARAISON	eq	(v1, v2)	tuple [int, int] tuple [real, real] tuple [bool, bool] tuple [char, char]	bool bool bool bool	
	neq	(v1, v2)	tuple [int, int] tuple [real, real] tuple [bool, bool] tuple [char, char]	bool bool bool bool	
	less	(v1, v2)	tuple [int, int] tuple [real, real] tuple [char, char]	bool bool bool	less(ent v1, ent v2)
	leq	(v1, v2)	tuple [int, int] tuple [real, real] tuple [char, char]	bool bool bool	leq(ent v1, ent v2)
	grt	(v1, v2)	tuple [int, int] tuple [real, real] tuple [char, char]	bool bool bool	grt(ent v1, ent v2)
	geq	(v1, v2)	tuple [int, int] tuple [real, real] tuple [char, char]	bool bool bool	geq(ent v1, ent v2)

Type	Opérateur	Catégorie de l'opérande	Mode de l'opérande	Mode du résultat	Description de l'opération et du résultat
D'ENTREE-SORTIE	iread	(v1, v2)	tuple [int, int]	int	- lecture sur l'unité v1 de v2 chiffres - résultat : l'entier représenté par ces chiffres.
	rread	(v1, v2, v3)	tuple [int, int, int]	real	- lecture sur v1 de deux suites de v2 et de v3 chiffres respectivement. - résultat : le real représenté par ces deux suites (partie entière et partie décimale).
	bread	v1	int	bool	- lecture sur v1 d'un booléen, résultat de l'opération
	cread	v1 (v1, v2)	int tuple [int, int]	char tuple [char, ..., char]	- lecture sur v1 d'un caractère, résultat de l'opération - lecture sur v1 d'une chaîne de v2 caractères, résultat de l'opération
	write	(v1, v2)	tuple [int, int] tuple [int, real] tuple [int, bool] tuple [int, char] tuple [int, tuple [char, ..., char]]	int real bool char tuple [char, ..., char]	écriture sur v1 de la valeur v2 (sous forme décimale pour les nombres)

. OPERATIONS_SUR_FICHIERS : NE DONNENT PAS DE RESULTAT

Type	Opérateur	Catégorie de l'opérande	Mode de l'opérande	Mode du résultat	Description de l'opération et du résultat
SUR FICHIERS	input	(v1, v2, v3)	tuple [int, tuple [char, ..., char] , proc none]	none	ouverture en entrée sur l'unité v1 du fichier identifié par v2. Appel de v3 en cas de fin de fichier.
	output	(v1, v2, v3)	tuple [int, tuple [char, ..., char], proc none]	none	idem, mais ouverture en sortie.
	close	(v1, v2, v3)	tuple [int, tuple [char, ..., char], proc none]	none	idem, mais fermeture en entrée ou en sortie.

APPENDICE III

TRAITEMENT DES DECLARATIONS :

tableaux faits à la compilation

Dans les exemples qui suivent, on présente d'abord l'état intermédiaire des tableaux après exécution de la première étape du traitement des déclarations (pages III.3 à III.6), puis l'état final de ces tableaux, après exécution de la deuxième étape (pages III.7 à III.8). On remarquera, en comparant ces deux états, que toutes les équivalences ont été découvertes, c'est-à-dire qu'il n'y a jamais deux représentations équivalentes dans les parties définitives, et que certaines réductions ont été effectuées.

Afin de lire ces tableaux, l'entier n contenu dans :

la colonne MODE de V-D,
ou la colonne MODE de M-D,
ou la colonne LISTE-MODE de C-L,
ou la colonne COMPOSANT de C-S,
ou la colonne COMPOSEE de M-U.

doit être interprété de la façon suivante :

a) $n \in [1,5]$ indique un mode de type mb (int, real, bool, char, none).
b) $n \in [6,64]$ indique un mode de type ml rangé en partie définitive de C-L
c) $n \in [65, 128]$ indique un mode de type ml rangé en partie provisoire de C-L.
d) $n \in [257, 289]$ indique un mode de type mu rangé en partie définitive de M-U.
e) $n \in [290, 320]$ indique un mode de type mu rangé en partie provisoire de M-U.
f) $n \in [-64, -6]$ indique un mode de type m rangé dans M-D.
g) $n = -1$ indique une erreur.

Enfin, il faut noter que dans l'état des tableaux après exécution de la deuxième étape, les possibilités c) et e) n'existent pas, car il n'y a plus de partie provisoire, et la possibilité f) n'existe pas non plus, car on connaît tous les modes représentés par les symboles qui ont pu être utilisés et on pointe directement vers les représentations de ces modes, sans passer par M-D, c'est-à-dire que l'on utilise seulement les possibilités a), b) ou d).

DECLARATIONS TRAITEES

```
(  'A' 'REP' 'UNION' .( 'INT' , 'BOOL' ). ,
P 'IS' 'UNION' .(  'PTR' 'INT'  ). ,
'M' 'REP' 'TUPLE' .( 'UNION' .( 'INT', 'A' ). , 'PTR' 'INT' ).,
'Z' 'REP' 'TUPLE' .( 'PTR' 'Z' , 'UNION' .( 'PTR' 'Z' , 'PTR' 'Z' ). ).,
[1] 'N1' 'REP' 'PTR' 'UNION' .( 'PTR'      'UNION' .( 'PTR' 'N1'    ,
[1] 'PTR' 'N2' ). , 'PTR' 'UNION' .( 'PTR' 'N1' ). ). ,
[1] 'N2'       'REP' 'PTR' 'UNION' .( 'PTR' 'UNION' .( 'PTR' 'N1' ). ). ,
[2] 'M1'     'REP' 'PTR' 'PTR' 'STRUCT' .(  'M1'     S). ,
[3] 'MODE1' 'REP' 'TUPLE' .( 'PTR' 'MODE2' , 'PTR' 'MODE1'  ). ,
[3] 'MODE2' 'REP' 'TUPLE' .( 'PTR' 'MODE1' , 'PTR' 'MODE2'  ). ,
[2] 'M2'     'REP' 'PTR' 'M3'   ,
[2] 'M3'     'REP' 'PTR' 'STRUCT' .(  'M3'     S ). ,
[4] 'B' 'REP' 'PTR' 'UNION' .( 'PTR' 'B' , 'PTR' 'UNION' .( 'PTR' 'B','B'
). ). 'IN'  (  )  ) &
          ↓    ↓    ↓
          [5]  [6]  [7]
```

COMMENTAIRES :

[1] Même cas que les modes présentés dans Ch.2, V.2., exemple 2.
[2] Les modes présentés dans Ch.2, IV.3., exemple 6.
[3] Les modes présentés dans Ch.2, IV.3., exemple 3.
[4] Même cas que le mode présenté dans Ch.2, V.2., exemple 1.
[5] in : fin de la partie déclarations.
[6] Partie d'instructions vide.
[7] Symbole final (dû aux besoins de l'implémentation).

.(et). représentent [et] respectivement ,
'identificateur' représente identificateur

RESULTATS DE LA PREMIERE ETAPE DU TRAITEMENT

TABLEAU V-0

	NOM DE LA VARIABLE	MODE	
6	.....	+ 0	→[1]
7	P	+ 258	

TABLEAU M-0

	NOM DU SYMBOLE	MODE	
6	.....	+ 0	→ [1]
7	A	+ 257	→ [2]
8	M	+ 7	→ [3]
9	Z	+ 65	→ [4]
10	N1	+ 69	
11	N2	+ 75	
12	M1	+ 78	
13	MODE1	+ 81	
14	MODE2	+ 84	
15	M2	+ 87	
16	M3	+ 88	
17	B	+ 90	

COMMENTAIRES :

[1] Pour le chaînage des niveaux de blocs.

[2] Pointeur vers M-U (partie définitive).

[3] Pointeur vers C-L (partie définitive).

[4] Pointeur vers C-L (partie provisoire).

TABLEAU C-L (PARTIE DEFINITIVE)

	.CONSTRUCTEUR.	LISTE_MODE
6	PTR	6
7	TUPLE	7

[1]

le Mode M

TABLEAU C-S (PARTIE DEFINITIVE)

	.SELECTEUR DE STRUCT.	COMPOSANT	SUIVANT
6		+ 1	0
7		+257	8
8		+ 6	0

[2] [3]

COMMENTAIRES :

[1] Pointeur vers C-S.

[2] Soit pointeur vers C-L, p.e. : 6 (ml) ;
soit pointeur vers M-U, p.e. : 257 (mu) ;
soit pointeur vers M-D, p.e. : -9 (md)
(voir C-L et C-S provisoires) ;
soit mb, p.e. 1.

[3] Soit pointeur vers C-S, p.e. : 8 ;
soit fin de liste : 0.

TABLEAU C-L (PARTIE PROVISOIRE)

	CONSTRUCTEUR	LISTE_MODE
65	TUPLE	129
66	PTR	130
67	PTR	132
68	PTR	133
69	PTR	134
70	PTR	135
71	PTR	136
72	PTR	137
73	PTR	138
74	PTR	139
75	PTR	140
76	PTR	141
77	PTR	142
78	PTR	143
79	PTR	144
80	STRUCT	145
81	TUPLE	146
82	PTR	147
83	PTR	149
84	TUPLE	150
85	PTR	151
86	PTR	153
87	PTR	154
88	PTR	155
89	STRUCT	156
90	PTR	157
91	PTR	158
92	PTR	159
93	PTR	160

TABLEAU C-S (PARTIE PROVISOIRE)

	SELECTEUR DE STRUCT.	COMPOSANT	SUIVANT
129		+ 66	131
130		- 9	0 → [1]
131		+290	0
132		- 9	0
133		- 9	0
134		+291	0
135		+292	0
136		- 10	0
137		- 11	0
138		+293	0
139		- 10	0
140		+294	0
141		+295	0
142		- 10	0
143		+ 79	0
144		+ 80	0
145	S	- 12	0
146		+ 82	148
147		- 14	0
148		+ 83	0
149		- 13	0
150		+ 85	152
151		- 13	0
152		+ 86	0
153		- 14	0
154		- 16	0
155		+ 89	0
156	S	- 16	0
157		+296	0
158		- 17	0
159		+297	0
160		- 17	0

COMMENTAIRES :

[1] Pointeur vers M-D (mode Z) (voir aussi les commentaires de la page III.4).

M-U (PARTIE DEFINITIVE)

LIGNE NO.	COMPOSEE DE	
257	+ 1	→ [4]
	+ 3	
258	+ 6	

[1] [2]

COMMENTAIRES :

[1] Ligne i de la matrice des unions.

[2] j de comp(j) de l'union i.

[3] Si j ∈ [-64, -6] : j ∈ I1, pointeur vers M-D, p.e. -17 (md).

[4] Si j ∈ [1,5] : j ∈ I2, p.e. 1(mb).

[5] Si j ∈ [6, 128] : j ∈ I3, pointeur vers C-L, p.e. 67 (ml).
Si j ∈ [257, 320] : j ∈ I4, pointeur vers M-U (mu).

M-U (PARTIE PROVISOIRE)

LIGNE NO.	COMPOSEE DE	
290	+ 67	→ [5]
	+ 68	
291	+ 70	
	+ 73	
292	+ 71	
	+ 72	
293	+ 74	
294	+ 76	
295	+ 77	
296	+ 91	
	+ 92	
297	- 17	→ [3]
	+ 93	

[1] [2]

RESULTATS DE LA DEUXIEME ETAPE DU TRAITEMENT

TABLEAU V-D

	NOM DE LA VARIABLE	MODE
6	.....	+ 0
7	P	+ 258

[1]

COMMENTAIRES :

[1] Le mode de P est mu.

[2] Le mode A est mu.

[3] à [10] modes ml.

[5] N1 ≐ N2.

[7] MODE1 ≐ MODE2.

[6], [8], [9] M1 ≠ M2.

[10] B est réduit.

TABLEAU M-D

	NOM DU SYMBOLE	MODE	
6	.....	+ 0	
7	A	+ 257	→ [2]
8	M	+ 7	→ [3]
9	Z	+ 8	→ [4]
10	N1	+ 10	↘
11	N2	+ 10	↗ [5]
12	M1	+ 13	→ [6]
13	MODE1	+ 16	↘
14	MODE2	+ 16	↗ [7]
15	M2	+ 18	→ [8]
16	M3	+ 19	→ [9]
17	B	+ 21	→ [10]

TABLEAU C-L (PARTIE DEFINITIVE)

	CONSTRUCTEUR	LISTE_MODE
6	PTR	6
7	TUPLE	7
8	TUPLE	9
9	PTR	10
10	PTR	12
11	PTR	13
12	PTR	14
13	PTR	15
14	PTR	16
15	STRUCT	17
16	TUPLE	18
17	PTR	19
18	PTR	21
19	PTR	22
20	STRUCT	23
21	PTR	24
22	PTR	25

TABLEAU C-S (PARTIE DEFINITIVE)

	SELECTEUR DE STRUCT.	COMPOSANT	SUIVANT
6		+ 1	0
7		+257	8
8		+ 6	0
9		+ 9	11
10		+ 8	0
11		+259	0
12		+260	0
13		+261	0
14		+ 10	0
15		+ 14	0
16		+ 15	0
17	S	+ 13	0
18		+ 17	20
19		+ 16	0
20		+ 17	0
21		+ 19	0
22		+ 20	0
23	S	+ 19	0
24		+262	0
25		+ 21	0

M-U (PARTIE DEFINITIVE)

LIGNE NO.	COMPOSEE DE
257	+ 1
	+ 3
258	+ 6
259	+ 9
260	+ 11
261	+ 12
262	+ 21
	+ 22

APPENDICE IV

L E P S E U D O-C O D E

On examinera dans l'ordre le pseudo-code correspond à :

A - unités (voir Ch.1, III)
- A.1. littéral
- A.2. variable
- A.3. expression-composée (voir aussi Ch. 1, I.2.3) : en incluant bloc et déclarations
- A.4. tuple
- A.5. procédure

B - expressions (voir Ch.1, V)
- B.1. appel de procédures
- B.2. choix d'éléments de séquences
- B.3. opérations (voir aussi App. II)
- B.4. séquences et boucles
- B.5. affectation

C - instructions de transfert (voir Ch.1, VI.2.1 et Ch.1, I.2.3)

D - opérations_sur_fichiers (voir Ch.1, VI.2.2)

E - transformation des valeurs (voir Ch.1, VIII et Ch.3, I)

F - manipulation de la pile d'exécution

La description est faite en présentant chaque pseudo-code, l'état de la pile d'exécution avant et après l'exécution du pseudo-code, et des commentaires dans certains cas.

La notation adoptée est la suivante :

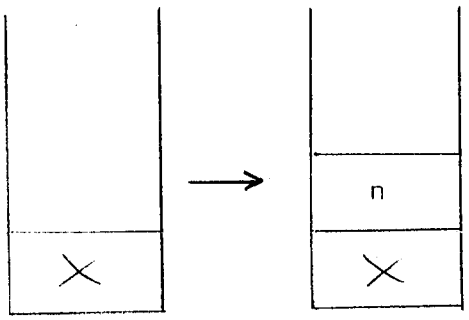
Symbole	Signification
n, n1, i, j	un entier
l, l1, l2, l3	entier indiquant la taille d'une valeur en unités de mémoire (voir Ch. 4, II).
r, r1, r2	un réel
b, b1, b2	un booléen
c, c1, c2, c3, ...	un caractère
$\underline{m}$, $\underline{n}$	un entier identifiant un mode, il est rangé dans 1u
v, v1, v2, v3	une valeur
$x_{\underline{m}}$	objet nil $\underline{m}$
α_P	adresse de la pile d'exécution
α_L, β_L	adresse de la mémoire libre
α_C, β_C	adresse où le pseudo-code est rangé
← ou PP	pointeur de la pile
<= ou PE	pointeur du début d'une expression_composée dans la pile

A - UNITES -

A.1 - Littéral -

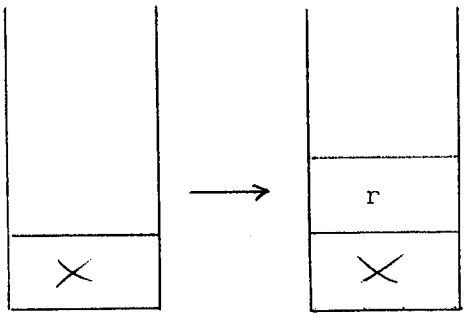
A.1.1 - Entier, réel, booléen, chaîne de caractères -

COPIE_INT d



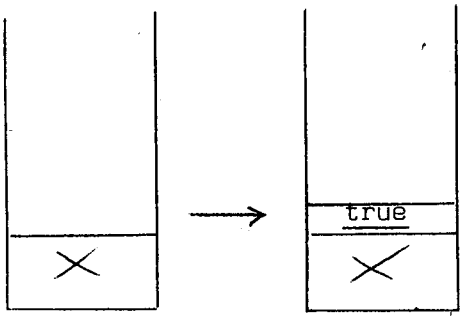
n = TAB_INT(d)
(voir Ch.4, III.5)

COPIE_REAL d

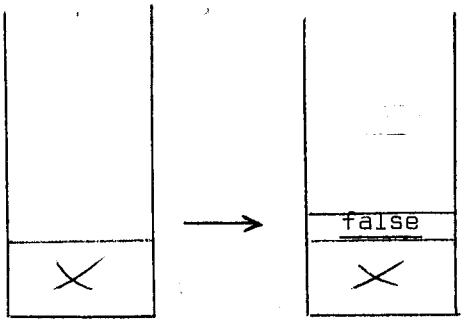


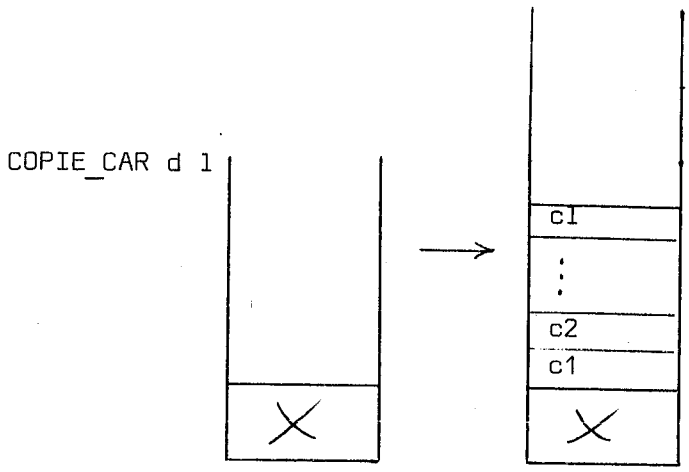
r = TAB_REAL(d)
(voir Ch.4, III.5)

COPIE_TRUE



COPIE_FALSE





c1 c2 ... cl : chaîne de caractères à partir de TAB_CAR(d) (voir Ch.4, III.5)

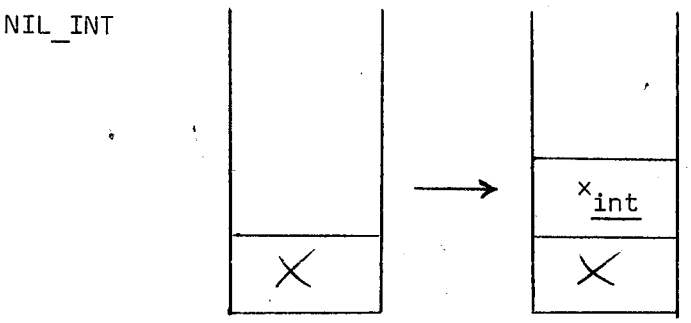
A.1.2 - Littéral nil mode -

Selon du mode du littéral on a les pseudo-code suivant :

NIL_INT, NIL_REAL, NIL_BOOL, NIL_CHAR 1, NIL_PTR,

NIL_SEQ, NIL_UNION, NIL_PROC.

Exemple :



x_{int} ≠ toute valeur entière représentable en BASEL.

Pour les littéraux de mode nil tuple [m1, ..., mn] et nil struct [m1 s1, ..., mn sn] on génère le pseudo-code correspondant à chaque élément de la valeur multiple.

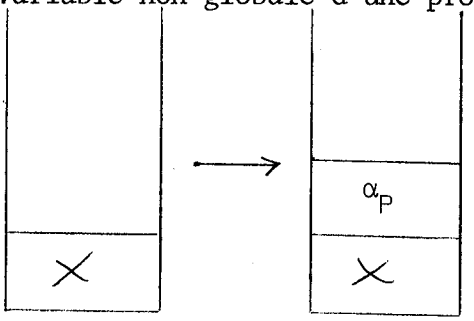
Ainsi pour nil tuple [real, ptr int] :

NIL_REAL
NIL_PTR

A.2 - Variable -

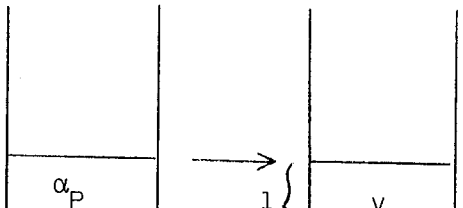
A.2.1 - Variable non globale d'une procédure -

LOC_ID n d



α_P = adresse déterminée par :
n = niveau du bloc où se trouve la variable.
n = 0 pour le bloc actuel,
n = 1 pour le bloc englobant,
n = 2 pour le bloc englobant l'englobant, etc....
d = déplacement où est rangée la valeur de la variable, relatif au début du bloc.

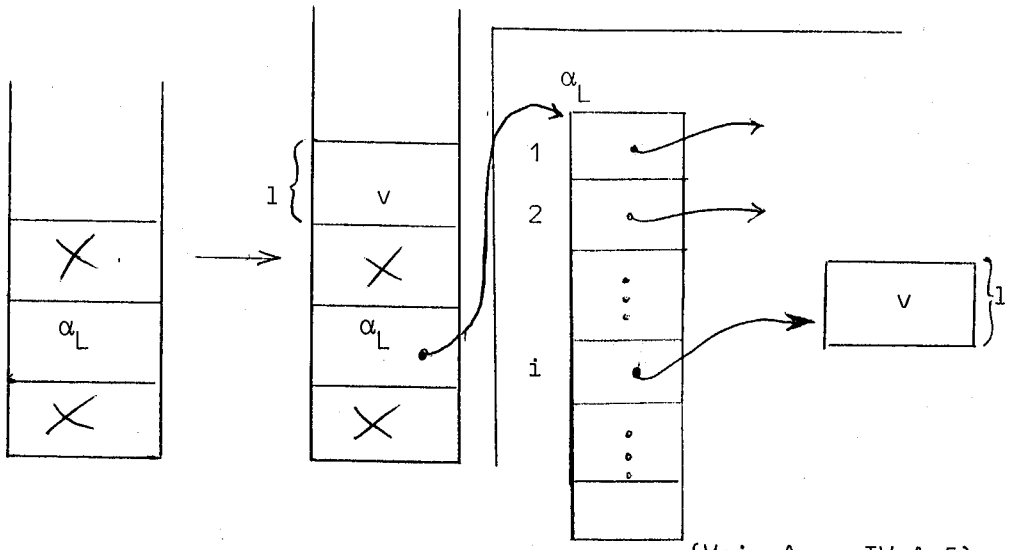
COPIE_ID l



v se trouve à l'adresse α_P.

A.2.2 - Variable globale d'une procédure -

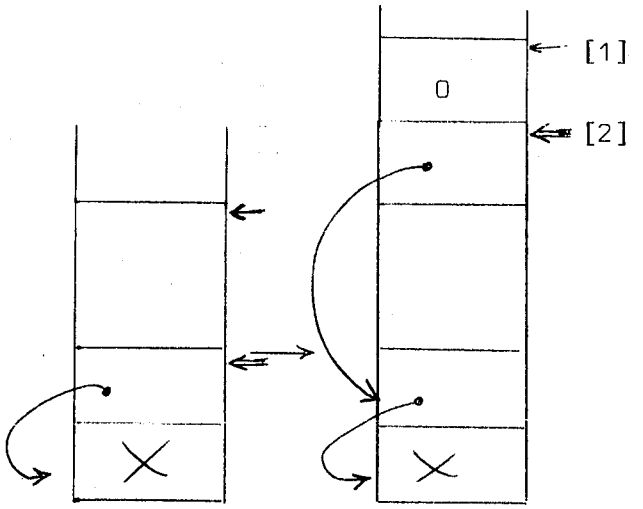
GLOBALE i l



(Voir App. IV,A.5)

A.3 - Expression composée -

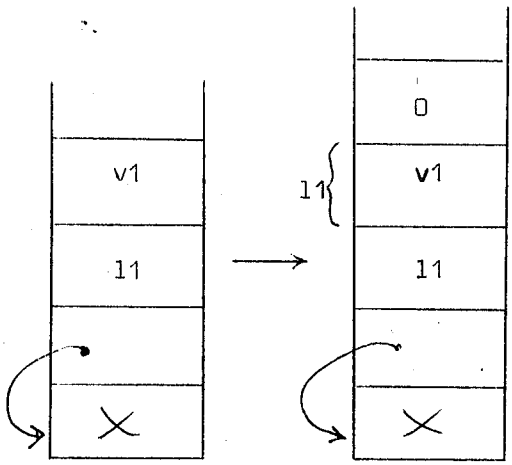
A.3.1 - Début d'une expression_composée -



[1] Pour la taille de la valeur de l'expression_ composée.

[2] Chaînage vers le bloc englobant.

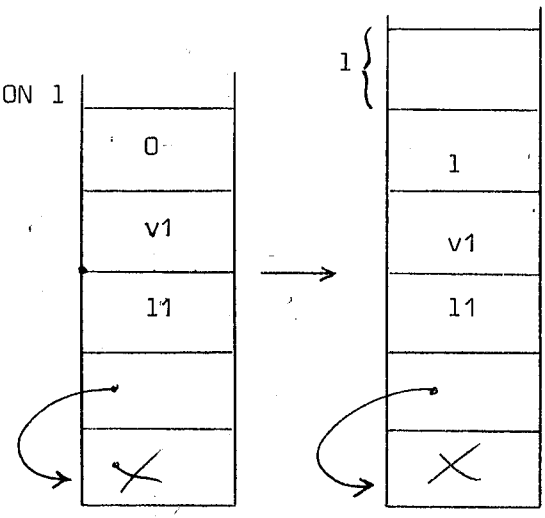
A.3.2 - Début d'un bloc -



[1] [1] Pour la taille de mémoire où sont rangées les variables du bloc dans la pile.

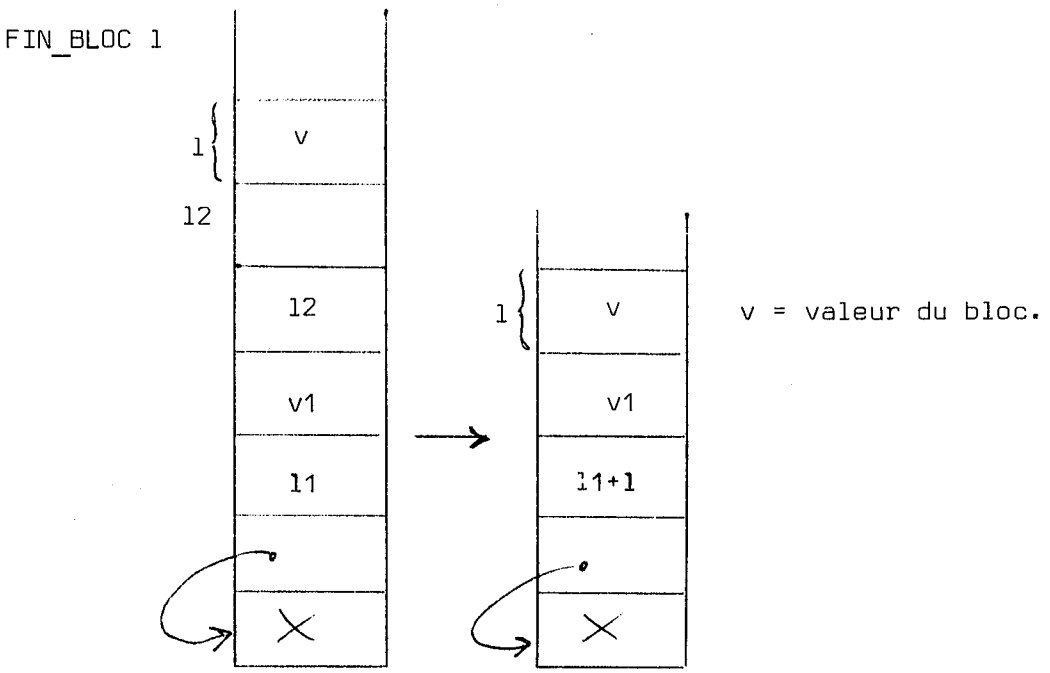
v1 = Valeur de l'expression_ composée au moment du début du bloc.

A.3.3 - Fin de la partie_déclarations -

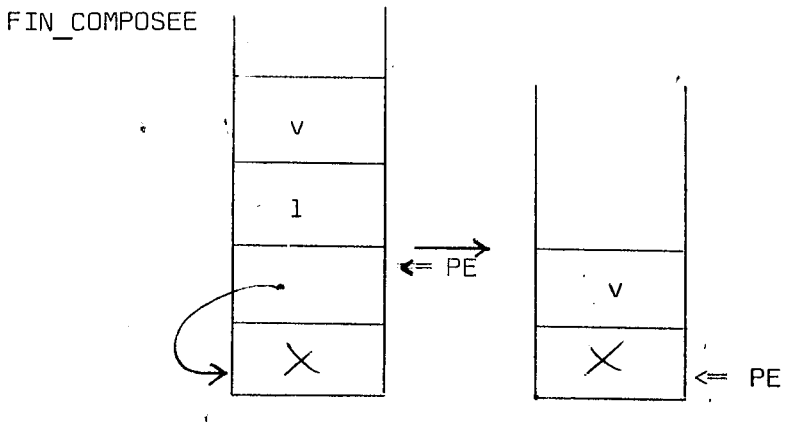


l = Σ ta(variable **déclarée**)

A.3.4 - Fin d'un bloc -



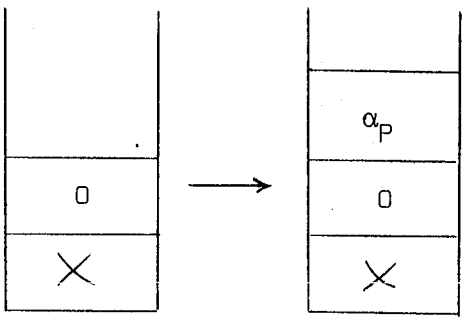
A.3.5 - Fin d'une expression_composée -



A.3.6 - Déclaration conditionnelle -

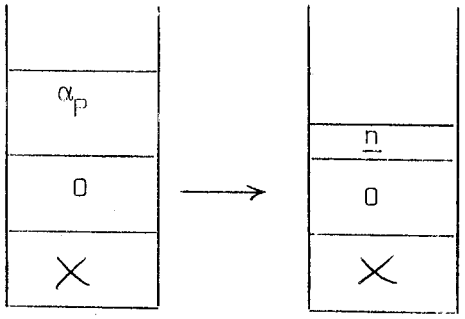
Le pseudo-code généré est le suivant :

LOC_ID n d



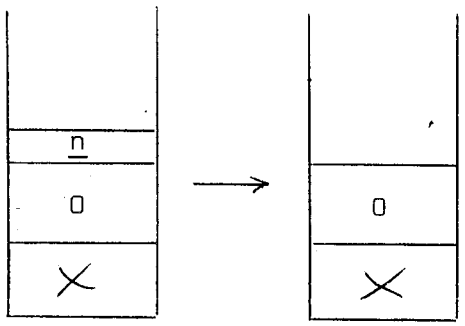
[1] défini par BLOC.

COPIE_MOD



n = indication de mode de la valeur courante de la variable de mode union rangée dans α_p.

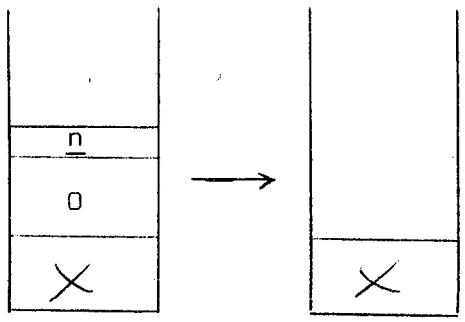
DCOND m α_C



si n = m.

Exécution du bloc.

ou



si n ≠ m

Exécution à partir d' α_C.

A.4 - Tuple -

Le traitement d'une tuple à la génération est identique à celui d'une expression_composée.

A.5 - Procédure -

La compilation d'une procédure produit :

pseudo-code

< ••• >

PILE α_C (voir App. IV. F)

ALLER 0 β_C (voir App. IV. C)

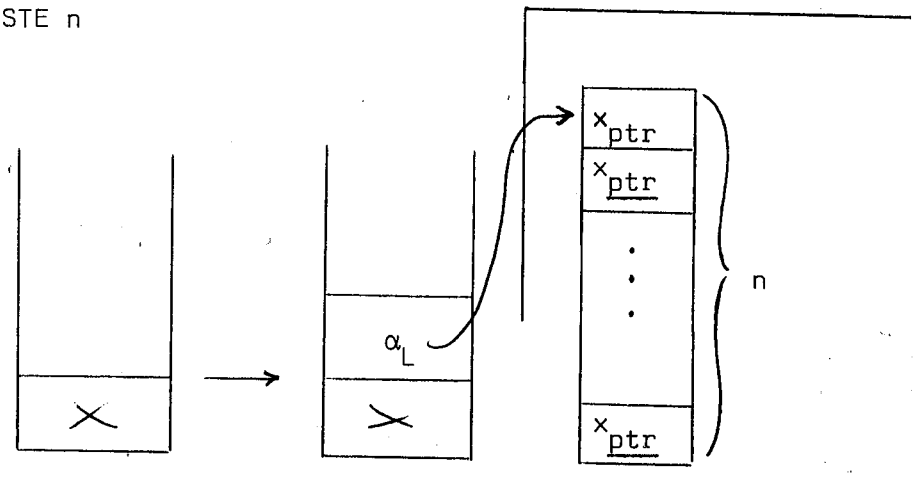
α_C — corps de la procédure.

β_C — copie de valeurs des variables globales.

A.5.1 - Copie de valeurs des variables globales -

a) - pour la liste des pointeurs vers les valeurs des variables globales

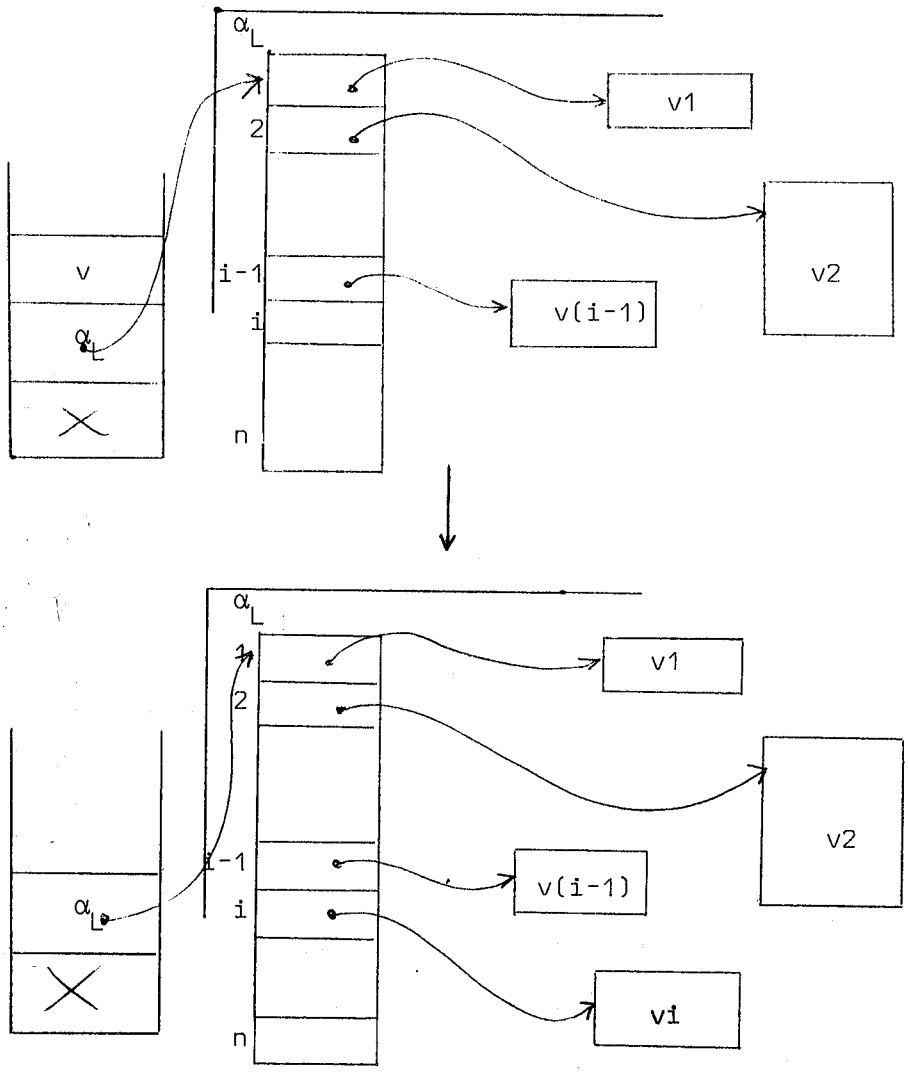
ALLOC_LISTE n



b) pour chaque variable :

LOC_ID **n d** (voir App. IV., A.2.1)
COPIE_ID 1 (voir App. IV., A.2.1)
ALLOC_VAR i 1

ALLOC_VAR i 1

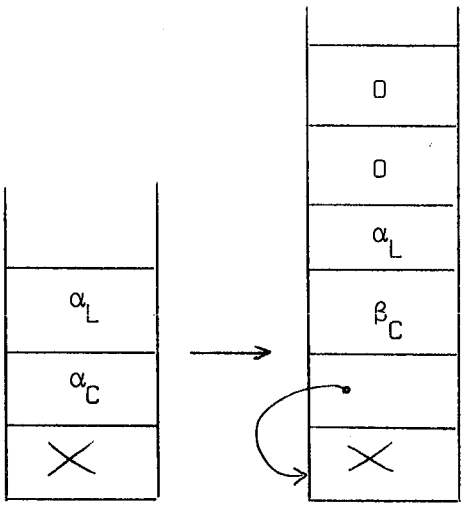


A.5.2 - Utilisation des variables globales dans le corps de la procédure : voir App. IV, A.2.2.).

B.1 - Appel de procédures

B.1.1 - Sans paramètres -

APPEL

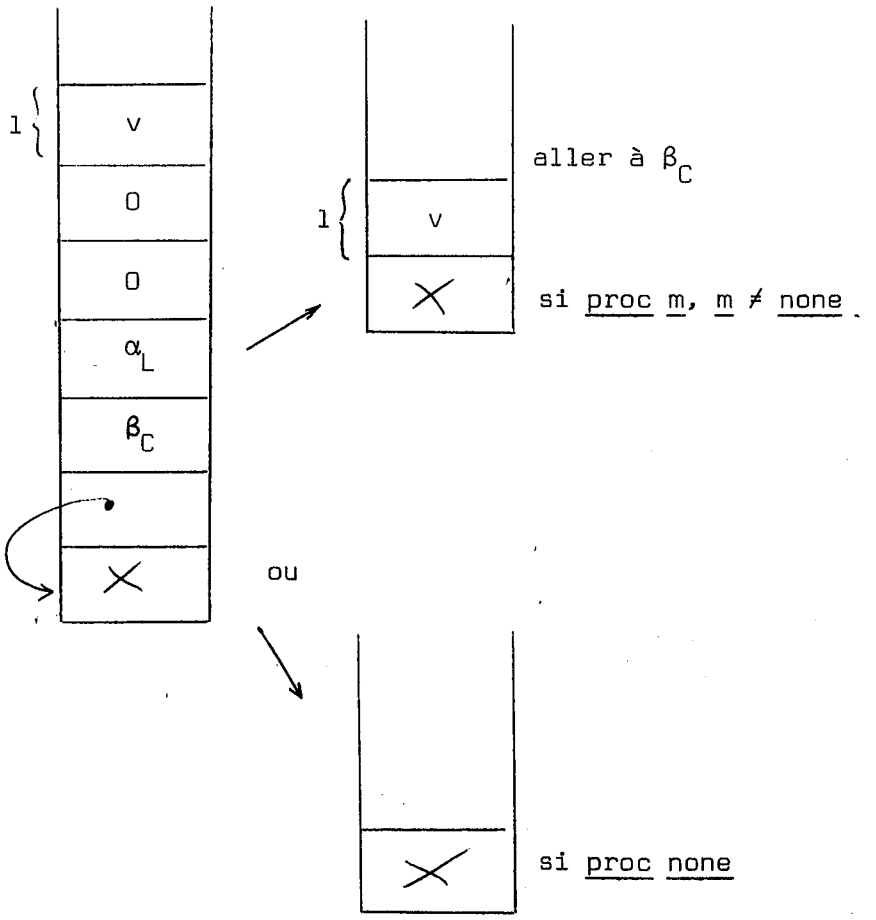


α_L = adresse de la liste de pointeurs vers les valeurs des variables globales.

α_C = adresse du corps de la procédure.

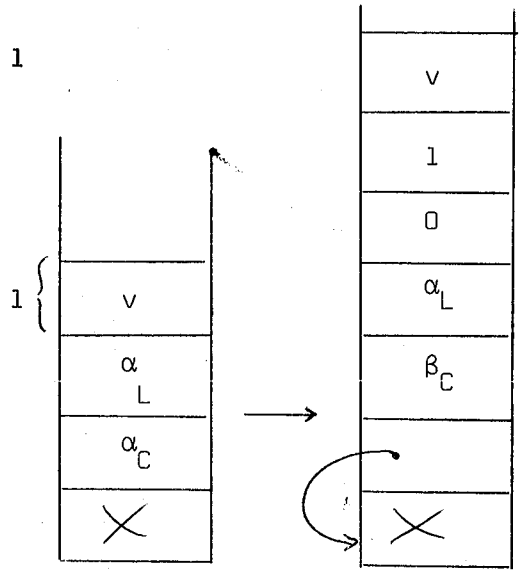
β_C = adresse de retour.

RETOUR (dans le corps de la procédure)



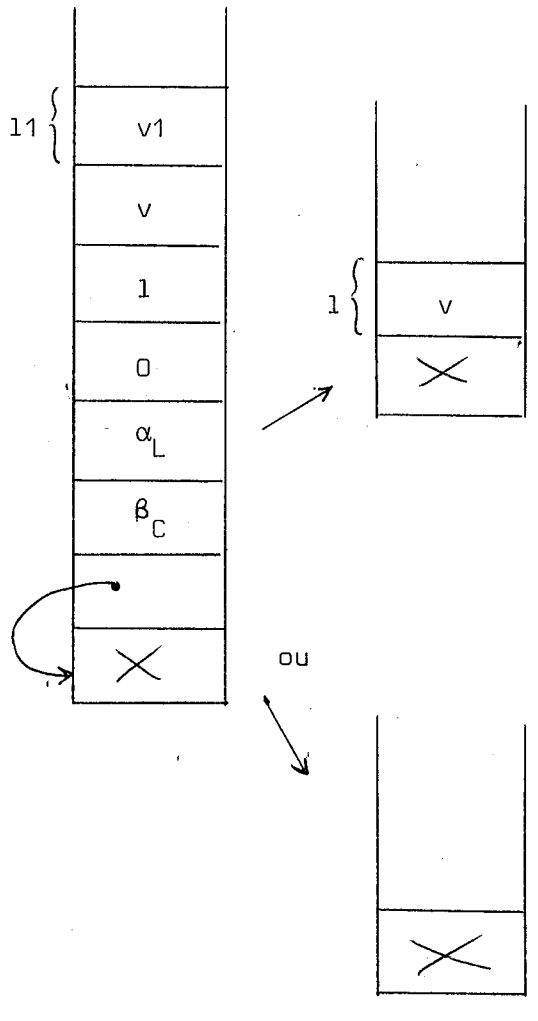
B.1.2 - Avec paramètres -

APPEL_PAR l



l = ta (paramètres)
v = valeur de la liste de paramètres
α_L = adresse de la liste de pointeurs vers les valeurs des variables globales
α_C = adresse du corps de la procédure
β_C = adresse de retour

RETOUR (dans le corps de la procédure)



v1 = valeur de la procédure

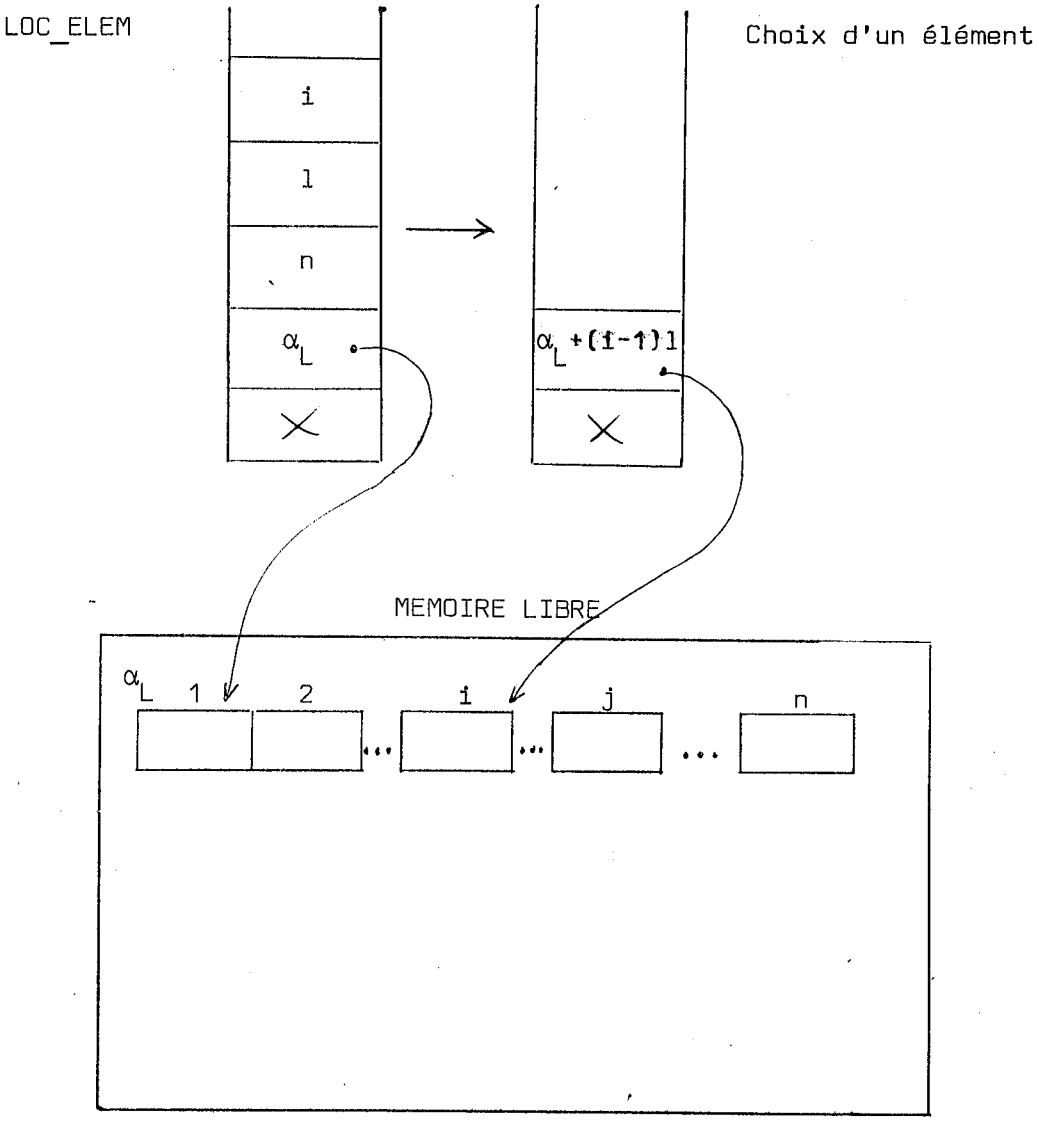
l1 = ta (v1)

aller à β_C

si proc [m1, ..., mn] m, m ≠ none

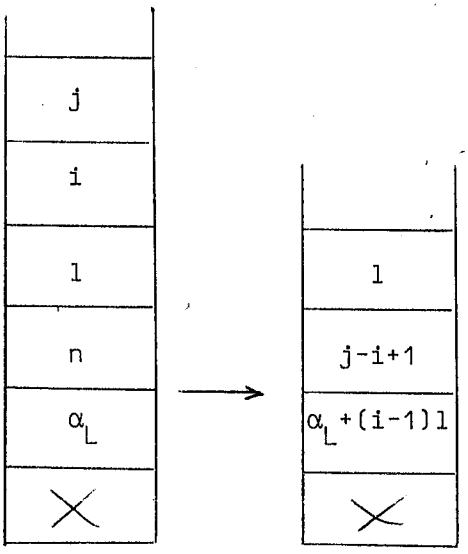
si proc [m1, ..., mn] none

B.2 - Choix d'éléments de séquences -



SOUS_SEQ

Choix d'une sous-séquence.



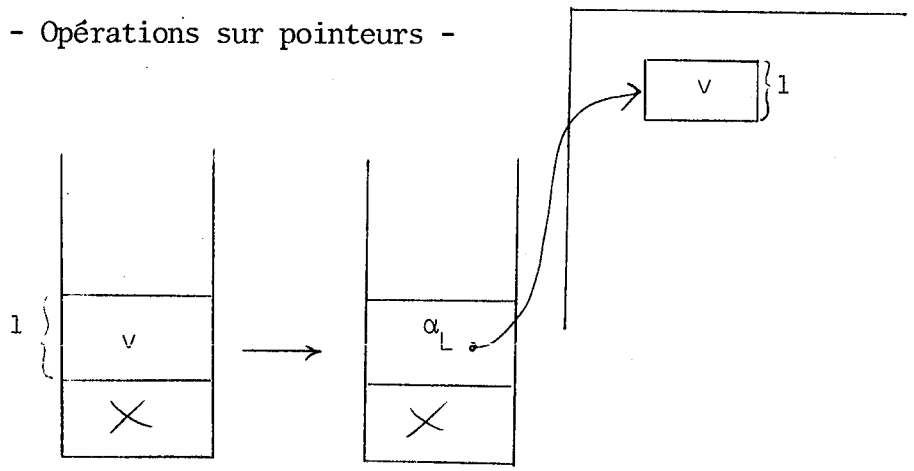
B.3 - Opérations qui calculent une valeur -

La valeur de l'opérante : v, (v1, v2) ou (v1, v2, v3) se trouve au somment de la pile. Le résultat remplace l'opérande après l'opération.

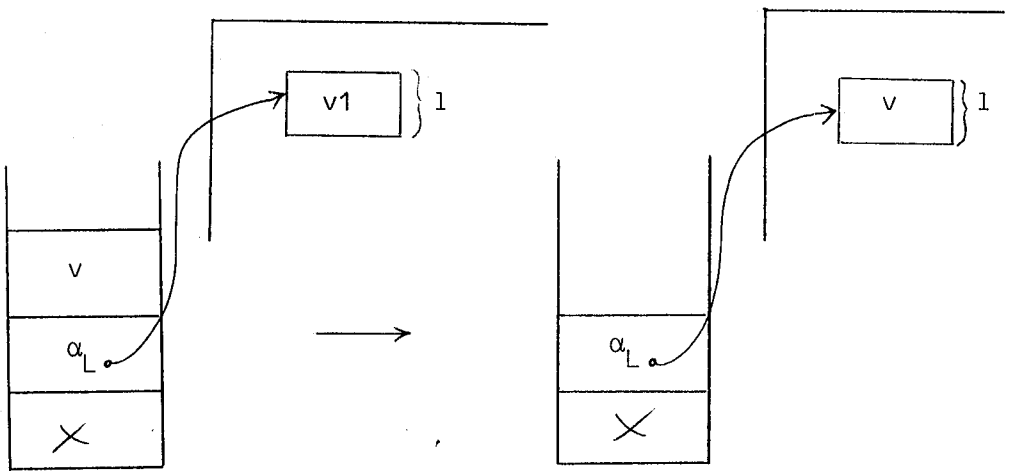
Le pseudo-code est classé d'après le type de l'opération.

B.3.1 - Opérations sur pointeurs -

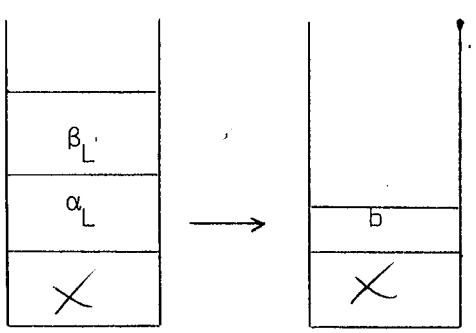
ALLOC 1



SET 1



SAME

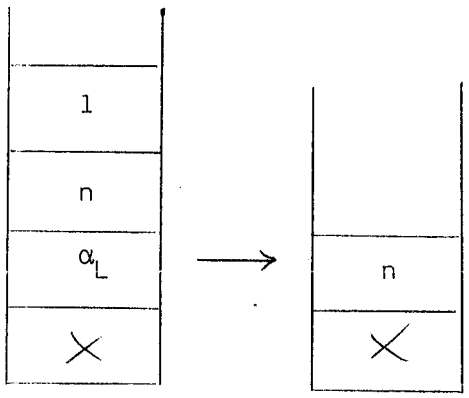


b = <u>true</u> si $\alpha_L = \beta_L$

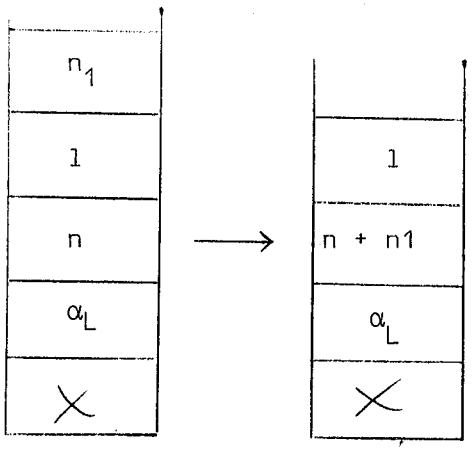
b = <u>false</u> si $\alpha_L \neq \beta_L$

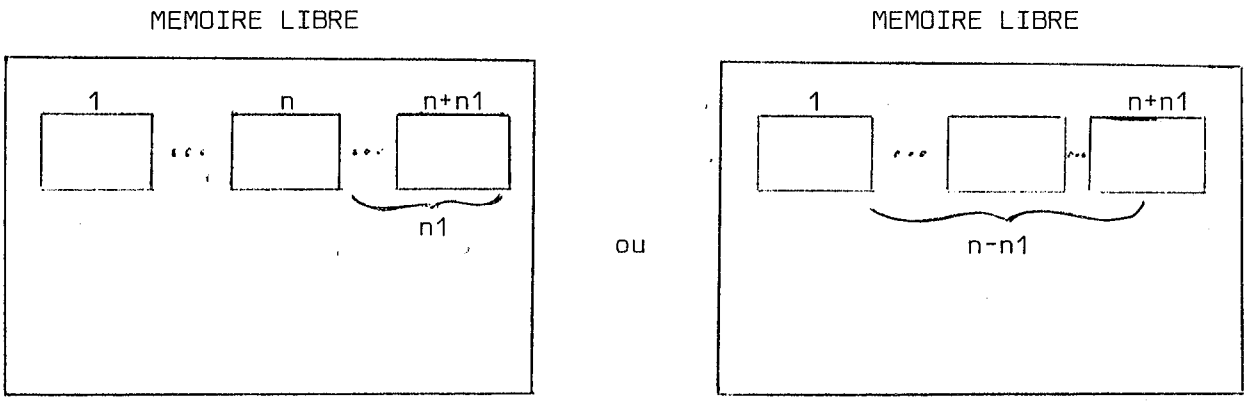
B.3.2 - Opérations sur séquences -

LENGTH



CHANGE





si n1>0

si n1<0

B.3.3 - Opérations arithmétiques, de conversion, booléennes, de comparaison, d'entrée-sortie.

Il s'agit du pseudo-code sans opérande.

Selon l'opération et le mode de l'opérande, on a les pseudo-code suivant :

Arithmétiques : IPLUS, RPLUS, IMINUS, RMINUS, IMUL, RMUL, IDIV, RDIV, IIDIV, REM, EXP, ABS, SIGN, ODD.

De conversion : ROUND, RENT, CENT, FLOAT.

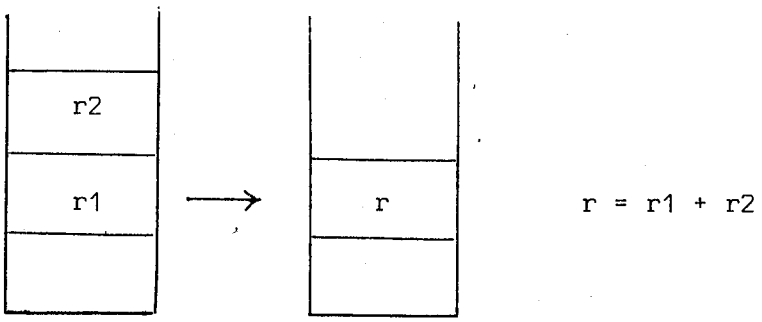
Booléens : OR, AND, NOT.

De comparaison : IEQ, REQ, BEQ, CEQ,
INEQ, RNEQ, BNEQ, CNEQ,
ILESS, RLESS, CLESS,
ILEQ, RLEQ, CLEQ,
IGRT, RGRT, CGRT,
IGRE, RGRE, CGRE.

D'entrée-sortie : IREAD, RREAD, BREAD, CREAD, WRITE.

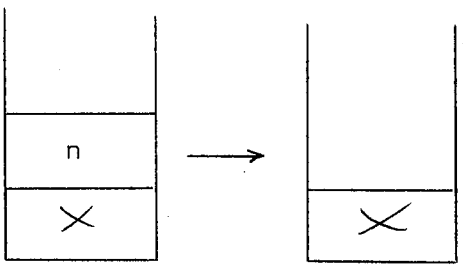
Exemple :

RPLUS



B.4 - Séquences et boucles -

BOUCLE α_C

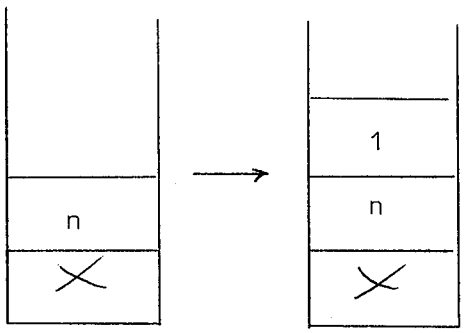


n = valeur qui précéde times

α_C = adresse de FIN_BOUCLE β_C

si n $\leq$ 0 ; aller à α_C + 2

ou

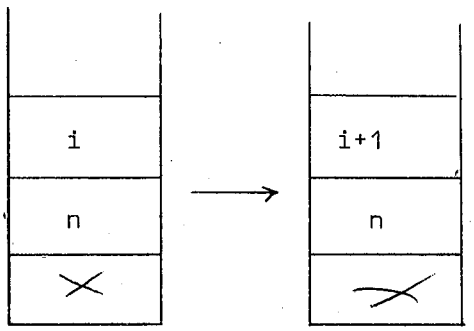


si n > 0 ; continuer en séquence

FIN_BOUCLE β_C

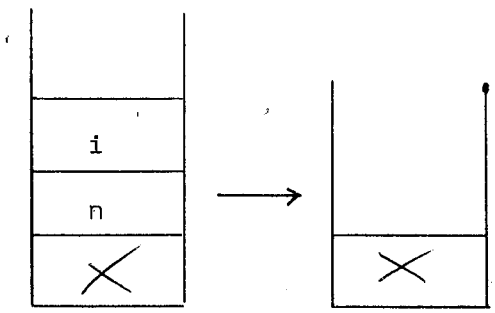
n = valeur qui précéde times

β_C = adresse de BOUCLE α_C



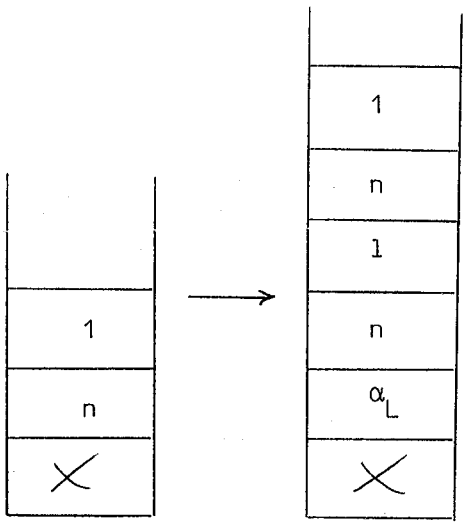
si i+1 $\leq$ n ; aller à β_C + 4

ou



si i+1 > n ; continuer en séquence, c'est-à-dire sortir de la boucle .

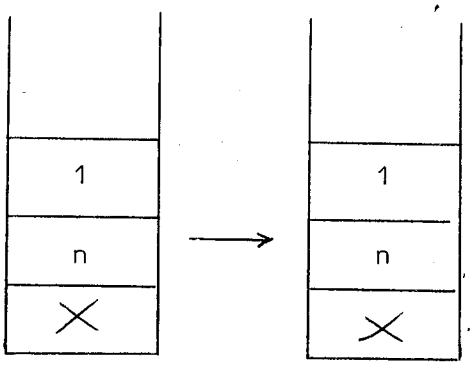
INIC_SEQ 1



α_L tel que :

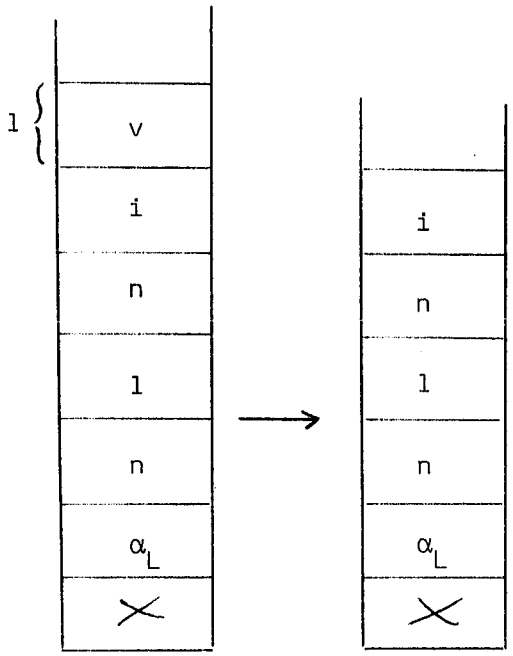
α_L, α_{L+1}, ... α_{L+nl-1} sont libres.

NOINIC_SEQ 1



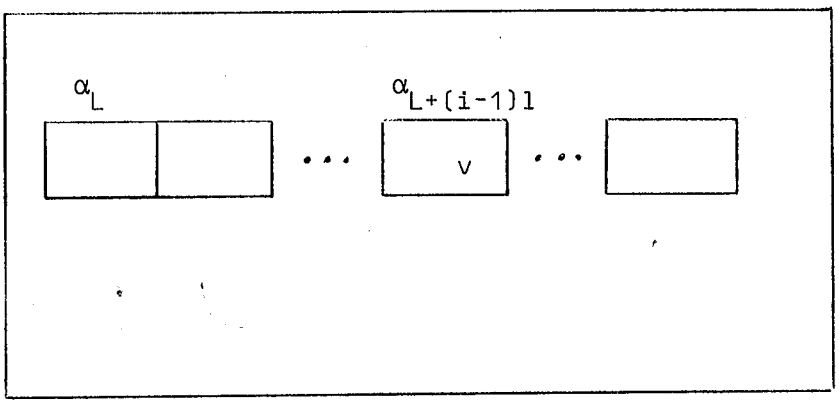
instruction vide qui remplace INIC_SEQ 1.
Si le mode de l'expression qui suit times est none.

ALLOC_SEQ 1



v = valeur de l'expression qui suit <u>times</u>.

l = ta (v)

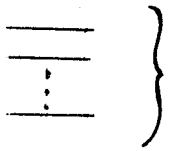


REMARQUE :

L'expression n <u>times</u> v ou mode de v ≠ <u>none</u> se traduit par

β_C - BOUCLE α_C

β_C+2 - INIC_SEQ 1



calcul de v

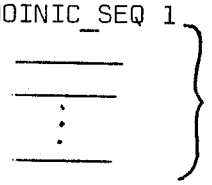
- ALLOC_SEQ 1

α_C - FIN_BOUCLE β_C

L'expression n <u>times</u> v où mode de v = <u>none</u> se traduit par :

β_C - BOUCLE α_C

NOINIC_SEQ 1

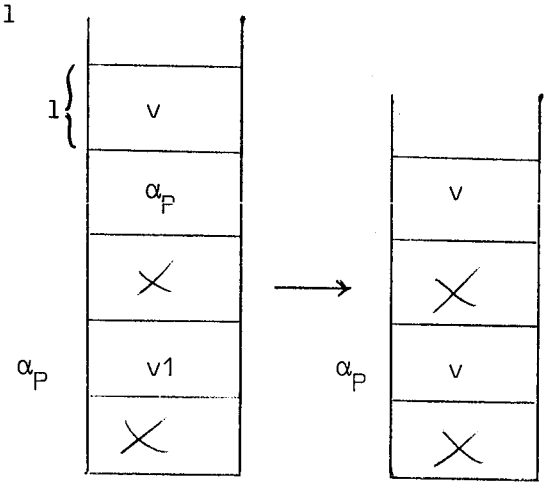


calcul de v

α_C - FIN_BOUCLE β_C

B.5 - Affectation -

AFFECT 1



α_P = emplacement dans la pile de la partie gauche d'une affectation.

REMARQUE : la compilation d'un nom généré : LOC_ID n d
(voir App. IV, A.2).

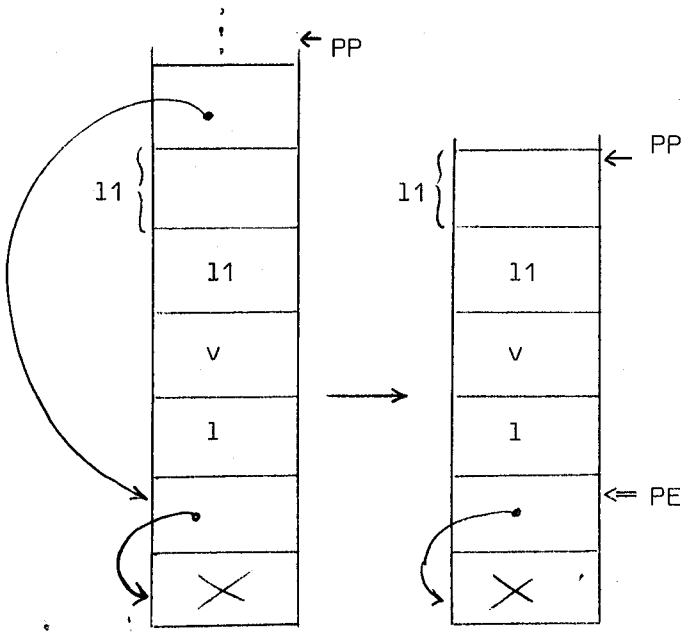
C - TRANSFERT -

C.1 - Instruction de transfert -

ALLER n α_C

- tant que n ≠ 0 : (PE ← PILE (PE) ; n ← n-1) ;
- définition de PP au niveau du début du bloc, après les mémoires où sont rangées les variables ;
- aller à α_C.

Exemple : si n = 1 :



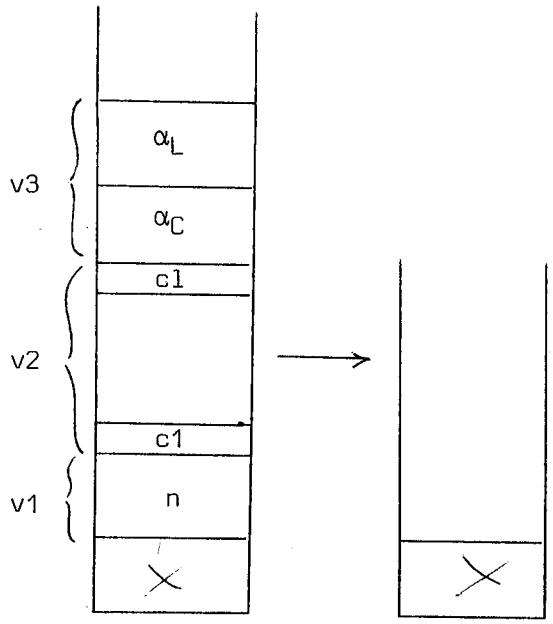
C.2 - Symboles → et |

Compilés à l'aide de ALLER n α_C et FIN_BLOC 1

D - OPERATIONS_SUR_FICHIERS

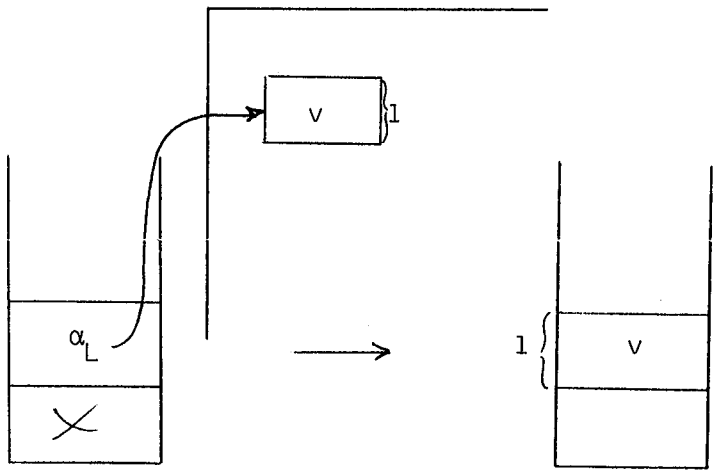
La valeur de l'opérande (v1, v2, v3) se trouve au sommet de la pile.

Pseudo-code : INPUT, OUTPUT, CLOSE

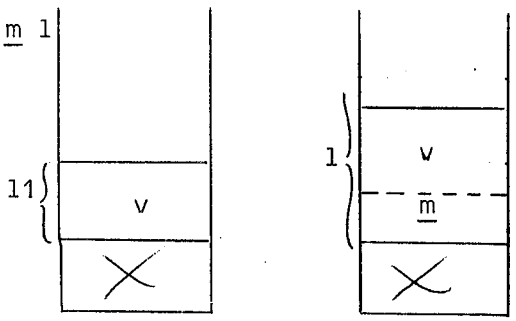


E - TRANSFORMATION DE VALEURS -

DEREP 1



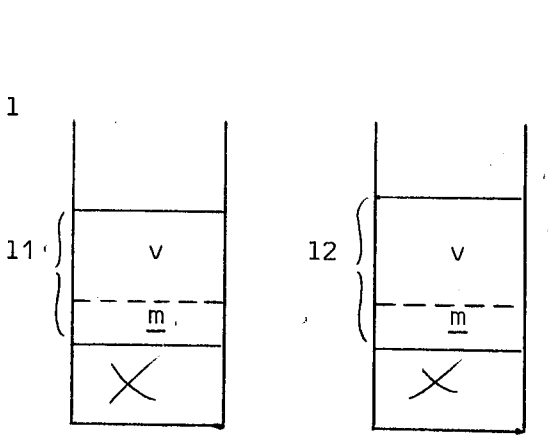
FORM_UNION m 1



m = mode de v

EXP_UNION 1

12 = 1+11



REMARQUE :

ST et DS : faites à l'aide des pseudo-code COPIE 1 d et DEPLACE 1 d (voir App. IV, E) afin de transformer les éléments de la valeur multiple.

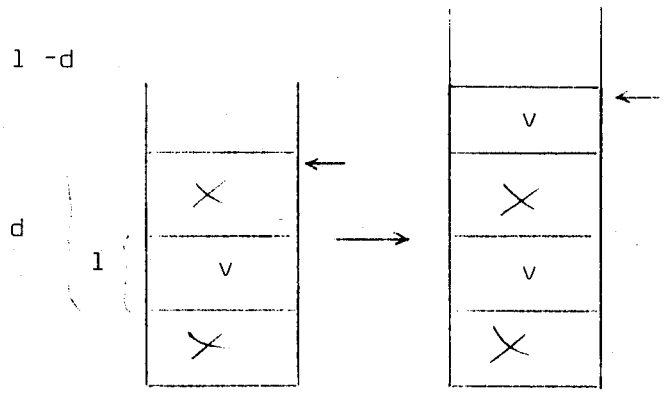
SP : faite à l'aide de POINTEUR n (voir App. IV, E) avec n < 0 et -n = ta(v), v = dernière valeur élaborée.

SQ : faite à l'aide du pseudo-code INIC_SEQ 1 et ALLOC_SEQ 1 (voir App. IV, B. 4).

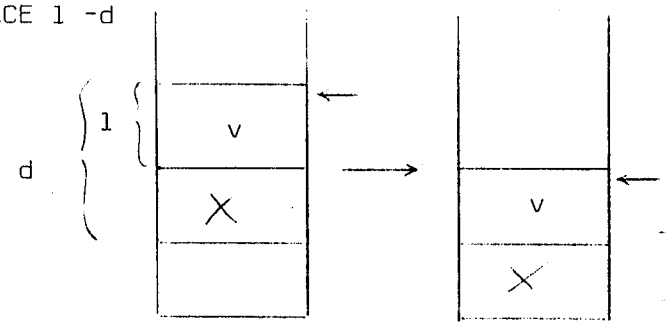
AP : APPEL (voir App. IV,B.1.1).

F - MANIPULATION DE LA PILE D'EXECUTION -

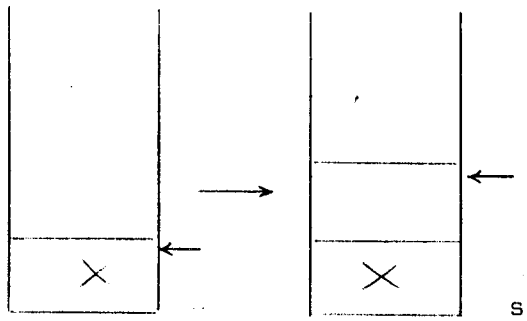
COPIE l -d



DEPLACE l -d

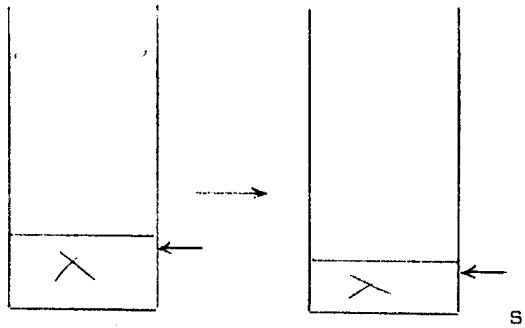


POINTEUR n



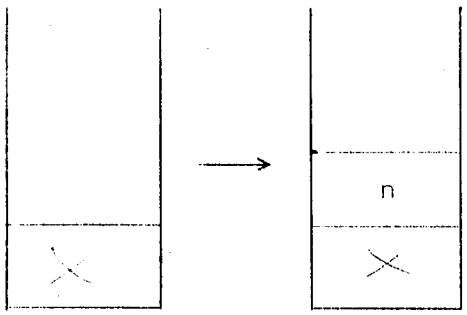
si n > 0

ou



si n < 0

PILE n



APPENDICE V

EXEMPLE DE GENERATION DU PSEUDO-CODE

(sortie du compilateur)

On présente la génération du pseudo-code au fur et à mesure du traitement (pages V.2 à V.7) et la liste totale du pseudo-code généré (pages V.8 à V.11). Les différences entre l'une et l'autre sont dues aux expressions dont la compilation a besoin de renseignements ultérieurs pour déterminer le pseudo-code à générer (p. e. dans le cas des boucles). A gauche du pseudo-code généré se trouvent les adresses α_C.

```
(P 'IS' 'PROC' 'PTR' 'PROC' 'PTR' 'STRUCT' .('INT' PI, 'PTR'
'BOOL' PB).,
T 'IS' 'TUPLE'.( 'UNION' .( 'INT' , 'REAL' )., 'BOOL' ). ,
PROSEQ 'IS' 'PROC' 'SEQ' 'REAL' ,
Z 'IS' 'STRUCT' .( 'INT' I, 'BOOL' B ). ,
BOOL1 'IS' 'BOOL' ,
I 'IS' 'INT',
J 'IS' 'INT' ,
S 'IS' 'PTR' 'SEQ' 'INT' ,
S1 'IS' 'SEQ' 'INT' ,
PPBOOL 'IS' 'MODO' ,
'MODO' 'REP' 'PROC' 'PTR' 'PROC' 'BOOL' ,
PTRBOOL 'IS' 'PTR' 'PTR' 'BOOL' ,
SINT 'IS' 'SEQ' 'INT' ,
SCHAR1 'IS' 'SEQ' 'CHAR' ,
SCHAR2 'IS' 'PTR' 'SEQ' 'CHAR' ,
SEQTUP 'IS' 'SEQ' 'TUPLE' .('INT' , 'BOOL')).,
A 'IS' 'REAL' ,
SET3 'IS' 'PTR' 'UNION' .( 'INT' ,'BOOL' ). ,
SET4 'IS' 'PROC' 'PTR' 'INT' ,
SAME1 'IS' 'PTR' 'PROC' 'INT' ,
SAME3 'IS' 'PTR' 'PROC' 'PTR' 'PROC' 'INT' ,
L 'IS' 'REAL'
'IN'
(Z 'IS' 'UNION' .('REAL' , 'TUPLE'.('INT' , 'SEQ' 'CHAR'). ,
'INT' , 'BOOL' ). 'IN'
Z = (A | (I= 'PLUS'.(I ,1).,(SCHAR1))|3));
```

```
6     LOC_ID      +   0   +   0   → pour Z
9     LOC_ID      + 104   +   2
12    COPIE_ID    +   4       }   → pour A
14    LOC_ID      +  28   +   3
17    LOC_ID      +  28   +   4
```

```
                20      COPIE_ID        +   4
                22      COPIE_INT       +   6
                24      PLUS
                25      AFFECT          +   4
                27      LOC_ID          +  76   +   4
                30      COPIE_ID        +  12
                32      COPIE_INT       +   7
                34      EXP_UNION       +   0
                36      AFFECT          +  17          → suppression
                38      POINTEUR        -  17
BJJL1  =  'NOT' ( 'OR' ( PPBOOL,PTRBOOL));

                40      LOC_ID          +  27   +   0
                43      LOC_ID          +  52   +   2
                46      COPIE_ID        +   8
                48      LOC_ID          +  60   +   2
                51      COPIE_ID        +   4
                53      COPIE           +   8   -  12
                56      APPEL
                57      DEREF           +   8
                59      APPEL
                60      COPIE           +   4   -   5
                63      DEREF           +   4
                65      DEREF           +   1
                67      DEPLACE         +   2   -  14
                70      OR
                71      NOT
                72      AFFECT          +   1
                74      POINTEUR        -   1
I= 'PLUS'('PLUS'(I,J),SINT(3));

                76      LOC_ID          +  28   +   0
                79      LOC_ID          +  28   +   2
                82      COPIE_ID        +   4
                84      LOC_ID          +  32   +   2
                87      COPIE_ID        +   4
                89      PLUS
                90      LOC_ID          +  64   +   1
                93      COPIE_ID        +  12
                95      COPIE_INT       +   7
                97      LOC_ELEM
                98      COPIE           +   4   -   8
               101      COPIE           +   4   -   8
               104      DEREF           +   4
               106      DEPLACE         +   8   -  16
               109      PLUS
               110      AFFECT          +   4
               112      POINTEUR        -   4
S.(2,5). (2);

               114      LOC_ID          +  36   +   0
               117      COPIE_ID        +   4
               119      DEREF           +  12
               121      COPIE_INT       +   8
               123      COPIE_INT       +   9
               125      SOUS_SEQ
               126      COPIE_INT       +   8
               128      LOC_ELEM
               129      DEREF           +   4
               131      POINTEUR        -   4
```

```
SCHAR2= 'ALLOC'  'SEQ'  'CHAR'  "ABCD"  ;

                133       LOC_ID          +  88   +   0
                136       COPIE_CAR       +   4   +   6
                139       INIC_SEQ        +   1
                141       POINTEUR        +   8
                143       COPIE           +   1   -  24
                146       ALLOC_SEQ       +   1
                148       COPIE           +   1   -  23
                151       ALLOC_SEQ       +   1
                153       COPIE           +   1   -  22
                156       ALLOC_SEQ       +   1
                158       COPIE           +   1   -  21
                161       ALLOC_SEQ       +   1
                163       POINTEUR        -   8
                165       DEPLACE         +  12   -  16
                168       ALLOC           +  12
                170       AFFECT          +   4
                172       DEREP           +  12
                174       POINTEUR        -  12
T=P;
                176       LOC_ID          +   8   +   0
                179       LOC_ID          +   0   +   0
                182       COPIE_ID        +   8
                184       APPEL
                185       DEREP           +   8
                187       APPEL
                188       DEREP           +   8
                190       COPIE           +   4   -   8
                193       FORM_UNION      +   1   +   1
                196       COPIE           +   4   -   9
                199       DEREP           +   1
                201       DEPLACE         +   6   -  14
                204       AFFECT          +   6
                206       POINTEUR        -   6
'SEQ'  'INT'  (1,2,'ALLOC'  3);

                208       COPIE_INT       +   6
                210       COPIE_INT       +   8
                212       COPIE_INT       +   7
                214       ALLOC           +   4
                216       INIC_SEQ        +   4
                218       POINTEUR        +   8
                220       COPIE           +   4   -  32
                223       ALLOC_SEQ       +   4
                225       COPIE           +   4   -  28
                228       ALLOC_SEQ       +   4
                230       COPIE           +   4   -  24
                233       DEREP           +   4
                235       ALLOC_SEQ       +   4
                237       POINTEUR        -   8
                239       DEPLACE         +  12   -  24
                242       POINTEUR        -  12
S1(3)  'TIMES' (P;P);

                244       LOC_ID          +  40   +   0
                247       COPIE_ID        +  12
                249       COPIE_INT       +   7
                251       LOC_ELEM
```

→ le DR n'est pas nécessaire. L'algorithme de la suppress est remplacé par (DR*AP*)* afin de simplifier l'implémentation.

```
                252       DEREP          +   4
                254       BOUCLE         + 254          → opérande provisoire, voir
                256       INIC_SEQ       +   1            pseudo-code définitif à
                258       LOC_ID         +   0    +   1  l'adresse 254, page V.9.
                261       COPIE_ID       +   8
                263       APPEL
                264       DEREP          +   8
                266       APPEL
                267       DEREP          +   8
                269       POINTEUR       -   8
                271       LOC_ID         +   0    +   1
                274       COPIE_ID       +   8
                276       ALLOC_SEQ      +   8
                278       FINBOUCLE      + 254
                280       POINTEUR       -  12
'PLUS'  (I,I)  'TIMES'  (2,());

                282       LOC_ID         +  28    +   1
                285       COPIE_ID       +   4
                287       LOC_ID         +  28    +   1
                290       COPIE_ID       +   4
                292       PLUS
                293       BOUCLE         + 293
                295       INIC_SEQ       +   1
                297       COPIE_INT      +   8
                299       ALLOC_SEQ      +   4
                301       FINBOUCLE      + 293
                303       POINTEUR       -  12
'TIMES'  'PLUS'  (I,3)  'TIMES' ('PLUS'(I,3);( ));

                305       COPIE_INT      +  10
                307       BOUCLE         + 307
                309       INIC_SEQ       +   1          → à changer par NOINIC-SEQ,
                311       LOC_ID         +  28    +   1  voir pseudo-code définitif
                314       COPIE_ID       +   4          à l'adresse 309, page V.10
                316       COPIE_INT      +   7
                318       PLUS
                319       BOUCLE         + 319
                321       INIC_SEQ       +   1
                323       LOC_ID         +  28    +   2
                326       COPIE_ID       +   4
                328       COPIE_INT      +   7
                330       PLUS
                331       POINTEUR       -   4
                333       FINBOUCLE      + 319
                335       FINBOUCLE      + 307
                337       POINTEUR       +   0
:'LENGTH' ( 'SEQ' 'CHAR' .("A" ). ) ;

                339       LOC_ID         +  28    +   0
                342       COPIE_CAR      +   1    +  10
                345       INIC_SEQ       +   1
                347       POINTEUR       +   8
                349       COPIE          +   1    -  21
                352       ALLOC_SEQ      +   1
                354       POINTEUR       -   8
                356       DEPLACE        +  12    -  13
                359       LENGTH
                360       AFFECT         +   4
                362       POINTEUR       -   4
```

```
J=   'LENGTH' (PROSEQ) ;
                    364       LOC_ID          +  32    +   0
                    367       LOC_ID          +  14    +   1
                    370       COPIE_ID        +   8
                    372       APPEL
                    373       LENGTH
                    374       AFFECT          +   4
                    376       POINTEUR        -   4
'SET'   (SET3,1);

                    378       LOC_ID          + 108    +   1
                    381       COPIE_ID        +   4
                    383       COPIE_INT       +   6
                    385       COPIE           +   4    -   8
                    388       COPIE           +   4    -   8
                    391       DEPLACE         +   8    -  16
                    394       FORM_UNION      +   1    +   1
                    397       SET             +   5
                    399       DEREF           +   5
                    401       POINTEUR        -   5
'SET'   (SET3 ,SET4 ) ;

                    403       LOC_ID          + 108    +   1
                    406       COPIE_ID        +   4
                    408       LOC_ID          + 112    +   1
                    411       COPIE_ID        +   8
                    413       COPIE           +   4    -  12
                    416       COPIE           +   8    -  12
                    419       DEPLACE         +  12    -  24

 ERREUR NO.  285
'SAME'   ( SAME1   ,SAME3   )   ;                          → le mode de l'opérande
                                                             de set est incorrect.
                    422       LOC_ID          + 120    +   1
                    425       COPIE_ID        +   4
                    427       LOC_ID          + 124    +   1
                    430       COPIE_ID        +   4
                    432       COPIE           +   4    -   8
                    435       COPIE           +   4    -   8
                    438       DEPLACE         +   8    -  16
                    441       DEREF           +   8
                    443       APPEL
                    444       SAME
                    445       POINTEUR        -   1
'SAME'   (SAME3   ,SAME1   ) ;

                    447       LOC_ID          + 124    +   1
                    450       COPIE_ID        +   4
                    452       LOC_ID          + 120    +   1
                    455       COPIE_ID        +   4
                    457       COPIE           +   4    -   8
                    460       COPIE           +   4    -   8
                    463       DEPLACE         +   8    -  16
                    466       INTERVERTIR
                    467       DEREF           +   8
                    469       APPEL
                    470       SAME
                    471       POINTEUR        -   1
SEQTUP= 'LENGTH' S1 'TIMES' ( 'TUPLE' .( 'INT','BOOL' ).Z)   )   & → symbole final
                                                                   dû aux besoins de
                                                                   l'implémentation.
```

473	LOC_ID	+	92	+	0
476	LOC_ID	+	40	+	0
479	COPIE_ID	+	12		
481	LENGTH				
482	BOUCLE	+	482		
484	INIC_SEQ	+	1		
486	LOC_ID	+	22	+	1
489	COPIE_ID	+	5		
491	COPIE	+	4	-	5
494	COPIE	+	1	-	5
497	REPLACE	+	5	-	10
500	ALLOC_SEQ	+	5		
502	FINBOUCLE	+	482		
504	AFFECT	+	12		

```
  6   LOC_ID       +   0   +   0
  9   LOC_ID       +  104  +   2
 12   COPIE_ID     +   4
 14   LOC_ID       +   28  +   3
 17   LOC_ID       +   28  +   4
 20   COPIE_ID     +   4
 22   COPIE_INT    +   6
 24   IPLUS
 25   AFFECT       +   4
 27   LOC_ID       +   76  +   4
 30   COPIE_ID     +   12
 32   COPIE_INT    +   7
 34   EXP_UNION    +   0
 36   AFFECT       +   17
 38   POINTEUR     -   17
 40   LOC_ID       +   27  +   0
 43   LOC_ID       +   52  +   2
 46   COPIE_ID     +   8
 48   LOC_ID       +   60  +   2
 51   COPIE_ID     +   4
 53   COPIE        +   8   -   12
 56   APPEL
 57   DEREP        +   8
 59   APPEL
 60   COPIE        +   4   -   5
 63   DEREP        +   4
 65   DEREP        +   1
 67   DEPLACE      +   2   -   14
 70   OR
 71   NOT
 72   AFFECT       +   1
 74   POINTEUR     -   1
 76   LOC_ID       +   28  +   0
 79   LOC_ID       +   28  +   2
 82   COPIE_ID     +   4
 84   LOC_ID       +   32  +   2
 87   COPIE_ID     +   4
 89   IPLUS
 90   LOC_ID       +   64  +   1
 93   COPIE_ID     +   12
 95   COPIE_INT    +   7
 97   LOC_ELEM
 98   COPIE        +   4   -   8
101   COPIE        +   4   -   8
104   DEREP        +   4
106   DEPLACE      +   8   -   16
109   IPLUS
110   AFFECT       +   4
112   POINTEUR     -   4
114   LOC_ID       +   36  +   0
117   COPIE_ID     +   4
119   DEREP        +   12
121   COPIE_INT    +   8
123   COPIE_INT    +   9
125   SOUS_SEQ
126   COPIE_INT    +   8
```

```
128   LOC_ELEM
129   DEREP          +   4
131   POINTEUR       -   4
133   LOC_ID         +  88   +   0
136   COPIE_CAR      +   4   +   6
139   INIC_SEQ       +   1
141   POINTEUR       +   8
143   COPIE          +   1   -  24
146   ALLOC_SEQ      +   1
148   COPIE          +   1   -  23
151   ALLOC_SEQ      +   1
153   COPIE          +   1   -  22
156   ALLOC_SEQ      +   1
158   COPIE          +   1   -  21
161   ALLOC_SEQ      +   1
163   POINTEUR       -   8
165   DEPLACE        +  12   -  16
168   ALLOC          +  12
170   AFFECT         +   4
172   DEREP          +  12
174   POINTEUR       -  12
176   LOC_ID         +   8   +   0
179   LOC_ID         +   0   +   0
182   COPIE_ID       +   8
184   APPEL
185   DEREP          +   8
187   APPEL
188   DEREP          +   8
190   COPIE          +   4   -   8
193   FORM_UNION     +   1   +   1
196   COPIE          +   4   -   9
199   DEREP          +   1
201   DEPLACE        +   6   -  14
204   AFFECT         +   6
206   POINTEUR       -   6
208   COPIE_INT      +   6
210   COPIE_INT      +   8
212   COPIE_INT      +   7
214   ALLOC          +   4
216   INIC_SEQ       +   4
218   POINTEUR       +   8
220   COPIE          +   4   -  32
223   ALLOC_SEQ      +   4
225   COPIE          +   4   -  28
228   ALLOC_SEQ      +   4
230   COPIE          +   4   -  24
233   DEREP          +   4
235   ALLOC_SEQ      +   4
237   POINTEUR       -   8
239   DEPLACE        +  12   -  24
242   POINTEUR       -  12
244   LOC_ID         +  40   +   0
247   COPIE_ID       +  12
249   COPIE_INT      +   7
251   LOC_ELEM
252   DEREP          +   4
254   BOUCLE         + 278
256   INIC_SEQ       +   8
258   LOC_ID         +   0   +   1
261   COPIE_ID       +   8
```

```
263   APPEL
264   DEREP          +    8
266   APPEL
267   DEREP          +    8
269   POINTEUR       -    8
271   LOC_ID         +    0   +    1
274   COPIE_ID       +    8
276   ALLOC_SEQ      +    8
278   FINBOUCLE      + 254
280   POINTEUR       -   12
282   LOC_ID         +   28   +    1
285   COPIE_ID       +    4
287   LOC_ID         +   28   +    1
290   COPIE_ID       +    4
292   IPLUS
293   BOUCLE         + 301
295   INIC_SEQ       +    4
297   COPIE_INT      +    8
299   ALLOC_SEQ      +    4
301   FINBOUCLE      + 293
303   POINTEUR       -   12
305   COPIE_INT      +   10
307   BOUCLE         + 335
309   NOINIC_SEQ     +    0
311   LOC_ID         +   28   +    1
314   COPIE_ID       +    4
316   COPIE_INT      +    7
318   IPLUS
319   BOUCLE         + 333
321   NOINIC_SEQ     +    0
323   LOC_ID         +   28   +    2
326   COPIE_ID       +    4
328   COPIE_INT      +    7
330   IPLUS
331   POINTEUR       -    4
333   FINBOUCLE      + 319
335   FINBOUCLE      + 307
337   POINTEUR       +    0
339   LOC_ID         +   28   +    0
342   COPIE_CAR      +    1   +   10
345   INIC_SEQ       +    1
347   POINTEUR       +    8
349   COPIE          +    1   -   21
352   ALLOC_SEQ      +    1
354   POINTEUR       -    8
356   DEPLACE        +   12   -   13
359   LENGTH
360   AFFECT         +    4
362   POINTEUR       -    4
364   LOC_ID         +   32   +    0
367   LOC_ID         +   14   +    1
370   COPIE_ID       +    8
372   APPEL
373   LENGTH
374   AFFECT         +    4
376   POINTEUR       -    4
378   LOC_ID         + 108   +    1
381   COPIE_ID       +    4
383   COPIE_INT      +    6
385   COPIE          +    4   -    8
```

```
388   COPIE          +    4   -    8
391   DEPLACE        +    8   -   16
394   FORM_UNION     +    1   +    1
397   SET            +    5
399   DEREP          +    5
401   POINTEUR       -    5
403   LOC_ID         +  108   +    1
406   COPIE_ID       +    4
408   LOC_ID         +  112   +    1
411   COPIE_ID       +    8
413   COPIE          +    4   -   12
416   COPIE          +    8   -   12
419   DEPLACE        +   12   -   24
422   LOC_ID         +  120   +    1
425   COPIE_ID       +    4
427   LOC_ID         +  124   +    1
430   COPIE_ID       +    4
432   COPIE          +    4   -    8
435   COPIE          +    4   -    8
438   DEPLACE        +    8   -   16
441   DEREP          +    8
443   APPEL
444   SAME
445   POINTEUR       -    1
447   LOC_ID         +  124   +    1
450   COPIE_ID       +    4
452   LOC_ID         +  120   +    1
455   COPIE_ID       +    4
457   COPIE          +    4   -    8
460   COPIE          +    4   -    8
463   DEPLACE        +    8   -   16
466   INTERVERTIR
467   DEREP          +    8
469   APPEL
470   SAME
471   POINTEUR       -    1
473   LOC_ID         +   92   +    0
476   LOC_ID         +   40   +    0
479   COPIE_ID       +   12
481   LENGTH
482   BOUCLE         +  502
484   INIC_SEQ       +    5
486   LOC_ID         +   22   +    1
489   COPIE_ID       +    5
491   COPIE          +    4   -    5
494   COPIE          +    1   -    5
497   DEPLACE        +    5   -   10
500   ALLOC_SEQ      +    5
502   FINBOUCLE      +  482
504   AFFECT         +   12
```

MODE RESULTANT=+0034

****