N° d'ordre : 329



# THÈSE

présentée à

l'UNIVERSITÉ DES SCIENCES ET TECHNIQUES
de LILLE I

pour obtenir le titre de

## DOCTEUR DE SPÉCIALITÉ

**(Mathématiques appliquées)**

par

Jean BUFFET

# ÉTUDES PRÉLIMINAIRES

# A LA COMPILATION D'ALGOL 68



THÈSE soutenue le 6 Juillet 1972

devant la Commission d'examen

**Monsieur P. BACCHUS**, Président
**Monsieur C. CARREZ**, examinateur
**Monsieur C. PAIR**, Examinateur
**Monsieur B. DRIEUX**, rapporteur

UNIVERSITE DES SCIENCES

ET TECHNIQUES DE LILLE.

## DOYENS HONORAIRES

MM. H.LEFEBVRE, PARREAU.

## PROFESSEURS HONORAIRES

M. ARNOULT, Mme BEAUJEU, MM. BEGHIN, BROCHARD, CAU, CHAPPELON, CHAUDRON, CORDONNIER, DEHEUVELS, DEHORNE, DEHORS, FAUVEL, FLEURY, P.GERMAIN, HEIM DE BALSAC, HOCQUETTE, KAMPE DE FERIET, KOURGANOFF, LAMOTTE, LELONG, Mme LELONG, MM. LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, NORMANT, PARISELLE, PASCAL, PAUTHENIER, PEREZ, ROIG, ROSEAU, ROUBINE, ROUELLE, WIEMAN, ZAMANSKY.

## PROFESSEURS TITULAIRES

| | | |
|---|---|---|
| M. | ANGRAND Jean Pierre | Géographie et Aménagement Spatial |
| M. | BACCHUS Pierre | Astronomie et Calcul |
| M. | BEAUFILS Jean Pierre | Chimie Générale |
| M. | BECART Maurice | I.U.T. Lille |
| M. | BLOCH Vincent | Psychophysiologie |
| M. | BIAYS Pierre | Géographie et Aménagement Spatial |
| M. | BONNEMAN Pierre | Chimie Industrielle |
| M. | BONTE Antoine | Géologie Appliquée |
| M. | BOUGHON Pierre | Mathématiques |
| M. | BOURIQUET Robert | Biologie Végétale |
| M. | CAPET Marcel-Francis | Institut de Préparation aux Affaires |
| M. | CELET Paul | Géologie Générale |
| M. | CONSTANT Eugène | Electronique |
| M. | CORSIN Pierre | Paléobotanique |
| M. | DECUYPER Marcel | Mathématiques |
| M. | DEDECKER Paul | Mathématiques |
| M. | DEFRETIN René | Biologie Animale - Directeur de l'Institut de Biologie Maritime de Wimereux |
| M. | DELATTRE Charles | Géologie Générale |
| M. | DURCHON Maurice | Biologie Animale |
| M. | FLATRES Pierre | Géographie et Aménagement Spatial |
| M. | FOURET René | Physique |
| M. | GABILLARD Robert | Electronique |
| M. | GEHU Jean Marie | Institut Agricole |
| M. | GLACET Charles | Chimie Organique |
| M. | GONTIER Gérard | Mécanique des Fluides |
| M. | GUILLAUME Jean | Biologie Végétale |
| M. | HEUBEL Joseph | Chimie Minérale |
| Mme | LENOBLE Jacqueline | Physique |
| M. | MONTREUIL Jean | Chimie Biologique |
| M. | POUZET Pierre | I.U.T. Lille |

| | |
|---|---|
| Mme SCHWARTZ Marie Hélène | Mathématiques |
| M. TILLIEU Jacques | Physique |
| M. TRIDOT Gabriel | Chimie Minérale Appliquée |
| M. VIDAL Pierre | Automatique |
| M. VIVIER Emile | Biologie Animale |
| M. WATERLOT Gérard | Géologie et Minéralogie |
| M. WERTHEIMER Raymond | Physique |

## PROFESSEURS A TITRE PERSONNEL

| | |
|---|---|
| M. BOUISSET Simon | Physiologie Animale |
| M. DELHAYE Michel | Chimie Physique et Minérale 1er Cycle |
| M. LEBRUN André | Electronique |
| M. LINDER Robert | Biologie Végétale |
| M. LUCQUIN Michel | Chimie Physique |
| M. PARREAU Michel | Mathématiques |
| M. PRUDHOMME Rémy | Sciences Economiques et Sociales |
| M. SAVARD Jean | Chimie Générale |
| M. SCHALLER François | Biologie Animale |
| M. SCHILTZ René | Physique |

## PROFESSEURS SANS CHAIRE

| | |
|---|---|
| M. BELLET Jean | Physique |
| M. BODARD Marcel | Biologie Végétale |
| M. BOILLET Pierre | Physique |
| M. DERCOURT Jean Michel | Géologie et Minéralogie |
| M. DEVRAINNE Pierre | Chimie Minérale |
| M. LOMBARD Jacques | Sciences Economiques et Sociales |
| Mle MARQUET Simone | Mathématiques |
| M. MONTARIOL Frédéric | Chimie Minérale Appliquée |
| M. PROUVOST Jean | Géologie et Minéralogie |
| M. VAILLANT Jean | Mathématiques |

## MAITRES DE CONFERENCES (et chargés de fonctions)

| | |
|---|---|
| M. ADAM Michel | Sciences Economiques et Sociales |
| M. ANDRE Charles | Sciences Economiques et Sociales |
| M. AUBIN Thierry | Mathématiques Pures |
| M. BEGUIN Paul | Mécanique des Fluides |
| M. BILLARD Jean | Physique |
| M. BKOUCHE Rudolphe | Mathématiques |
| M. BOILLY Bénoni | Biologie Animale |
| M. BONNEMAIN Jean Louis | Biologie Végétale |
| M. BONNOT Ernest | Biologie Végétale |
| M. BRIDOUX Michel | I.U.T. Béthune |
| M. BRUYELLE Pierre | Géographie et Aménagement Spatial |

| | |
|---|---|
| M. CAPURON Alfred | Biologie Animale |
| M. CARREZ Christian | Calcul |
| M. CHOQUET Marcel | I.U.T. Lille |
| M. CORDONNIER Vincent | Calcul |
| M. CORTOIS Jean | Physique |
| M. COULON Jean Paul | Electrotechnique |
| M. DEBRABANT Pierre | Sciences appliquées |
| M. ESCAIG Bertrand | Physique |
| Mme EVRARD Micheline | I.U.T. Lille |
| M. FAIDHERBE Jacques | Biologie Animale |
| M. FONTAINE Jacques | I.U.T. Lille |
| M. FROELICH Daniel | Sciences Appliquées |
| M. GAMBLIN André | Géographie et Aménagement Spatial |
| M. GOBLOT Rémi | Mathématiques |
| M. GOSSELIN Gabriel | Sciences Economiques et Sociales |
| M. GOUDMAND Pierre | Chimie Physique |
| M. GRANELLE | Sciences Economiques et Sociales |
| M. GRUSON Laurent | Mathématiques |
| M. GUIBAULT Pierre | Physiologie Animale |
| M. HERMAN Maurice | Physique |
| M. HUARD de la MARRE Pierre | Calcul |
| M. JOLY Robert | Biologie (Amiens) |
| M. JOURNEL Gérard | Sciences Appliquées |
| Mle KOSMANN Yvette | Mathématiques |
| M. LABLACHE-COMBIER Alain | Chimie Générale |
| M. LACOSTE Louis | Biologie Végétale |
| M. LANDAIS Jean | Chimie Organique |
| M. LAURENT François | Automatique |
| M. LAVAGNE Pierre | Sciences Economiques et Sociales |
| Mle LEGRAND Solange | Mathématiques |
| M. LEHMANN Daniel | Mathématiques |
| Mme LEHMANN Josiane | Mathématiques |
| M. LENTACKER Firmin | Géographie et Aménagement Spatial |
| M. LEROY Jean Marie | ENSCL |
| M. LEROY Yves | I.U.T. Lille |
| M. LHENAFF René | Géographie et Aménagement Spatial |
| M. LOCQUENEUX Robert | Physique |
| M. LOUAGE Francis | Sciences Appliquées |
| M. LOUCHEUX Claude | Chimie Physique |
| M. MAES Serge | Physique |
| Mme MAILLET Monique | Sciences Economiques et Sociales |
| M. MAIZIERES Christian | Automatique |
| M. MALAUSSENA Jean Louis | Sciences Economiques et Sociales |
| M. MESSELYN Jean | Physique |
| M. MIGEON Michel | Sciences Appliquées |
| M. MONTEL Marc | Physique |
| M. MONTUELLE Bernard | I.U.T. Lille |
| M. MUSSCHE Guy | Sciences Economiques et Sociales |
| M. NICOLE Jacques | E.N.S.C.L. |
| M. OUZIAUX Roger | Sciences Appliquées |
| M. PANET Marius | Electrotechnique |
| M. PAQUET Jacques | Sciences Appliquées |
| M. PARSY Fernand | Mécanique des Fluides |
| M. PONSOLLE Louis | Chimie (Valenciennes) |
| M. POVY Jean Claude | Sciences Appliquées |
| M. RACZY Ladislas | Radioélectricité |
| Mme RENVERSEZ Françoise | Sciences Economiques et Sociales |
| M. ROUSSEAU Jean Paul | Physiologie Animale |

| | |
|---|---|
| M. ROYNETTE Bernard | Mathématiques |
| M. SALMER Georges | Electronique |
| M. SEGUIER Guy | I.U.T. Béthune |
| M. SIMON Michel | Sciences Economiques et Sociales |
| M. SMET Pierre | Physique |
| M. SOMME Jean | Géographie et Aménagement Spatial |
| M. THOMAS Daniel | Chimie Minérale Appliquée |
| M. TOULOTTE Jean Marc | Sciences Appliquées |
| M. TREANTON Jean René | Sciences Economiques et Sociales |
| M. VANDORPE Bernard | Sciences Appliquées |
| M. VILETTE Michel | I.U.T. Béthune |
| M. WATERLOT Michel | Géologie Générale |
| Mme ZINN JUSTIN Nicole | Mathématiques. |

*C'est un grand plaisir pour moi d'exprimer toute ma gratitude à Monsieur le Professeur* Bacchus, *Directeur de l'UER Informatique-Electronique-Electrotechnique et Automatique, pour l'honneur qu'il me fait de présider le jury de cette Thèse.*

*Je remercie vivement Monsieur le Professeur* Carrez *et Monsieur* Pair, *Professeur à l'Université de Nancy, pour l'intérêt qu'ils portent à ce travail en acceptant bien volontiers de le juger.*

*Que Monsieur le Professeur* Drieux *qui m'a donné l'idée de ce travail, soit assuré de toute ma reconnaissance pour ses nombreux conseils et les encouragements qu'il n'a cessé de me prodiguer.*

*Je tiens également à remercier les membres du Groupe Algol 68 de l'AFCET, car ils m'ont permis d'approfondir ma connaissance du langage Algol 68.*

*Que Madame* Cantegrit, *qui s'est particulièrement attachée à rendre agréable la présentation de cette Thèse, Madame et Monsieur* Debock *ainsi que Monsieur* Soyez, *qui ont aussi contribué à sa réalisation matérielle, trouvent ici l'expression de mes plus sincères remerciements.*

*à mes Parents.*

# TABLE DES MATIERES

# INTRODUCTION

D'après le rapport [1] qui le définit, Algol 68 est un langage algorithmique conçu pour communiquer des algorithmes, pour les exécuter efficacement sur un éventail de calculateurs divers et pour faciliter leur enseignement à des étudiants. Dans ce travail nous ne nous préoccuperons pas de ce dernier point ; en revanche nous nous intéresserons principalement au deuxième de ces trois aspects, et plus particulièrement à la compilation du langage, nous ne négligerons pas pour autant le premier aspect puisque nous utiliserons Algol 68 pour décrire de façon concise un certain nombre d'algorithmes.

Au premier abord, une compilation est un processus de traduction d'un langage dans un autre. La traduction de textes écrits en une langue naturelle nécessite d'abord la reconnaissance des lettres utilisées ; cette identification peut poser des problèmes dans le cas de personnes qui écrivent mal (par exemple la différentiation des "u" et des "n" peut demander une analyse assez poussée du contexte pour lever les ambiguités qui en découlent). D'une manière analogue, en Algol 68, un même symbole peut être représenté de plusieurs façons et dans certains cas leur identification met en oeuvre des algorithmes assez complexes. Nous supposerons que ces problèmes sont résolus.

Les traducteurs réalisent ensuite, plus ou moins consciemment, une analyse grammaticale du texte à traduire en tenant compte des règles régissant l'écriture de ce texte dans une langue naturelle donnée. Cette analyse comprend une vérification des "règles d'accord". Dans le cas des langages de programmation la démarche est analogue ; cependant on a pris l'habitude de séparer plus nettement l'analyse en plusieurs phases : l'analyse de la structure telle qu'elle peut être décrite par un certain type de grammaires (c-grammaires), la recherche de ce que désignent certaines constructions (analogue à la recherche des antécédents des pronoms) selon des règles en général non formelles, la vérification de la compatibilité des "types".

En Algol 68, la grammaire formelle est originale et elle permet de décrire formellement les règles de compatibilité ; la recherche des "antécédents" est néanmoins décrite de manière informelle et comme la vérification des compatibilités est liée à cette recherche, l'exploitation directe de la grammaire

d'Algol 68 pour réaliser l'analyse n'est pas très efficace.

Enfin dans le cas de la traduction des langues naturelles il n'est pas nécessaire d'envisager de représenter matériellement, autrement que sur le papier, les objets ou les actions que décrit le texte. Pour un langage algorithmique la traduction est destinée à une machine incapable de se représenter par elle même les objets définis dans un programme ; d'autre part cette machine doit exécuter les actions à l'aide d'actions élémentaires qui sont décrites par un langage assez simple.

Nous ne traiterons pas de ce problème de la représentation interne des valeurs dans la mémoire de la machine, car il existe plusieurs ouvrages [8,9] où des solutions lui ont été proposées ; en ce qui concerne le passage du langage évolué au langage de la machine, les techniques utilisables sont classiques et les problèmes spécifiques qui se posent ne seront pas non plus abordés.

Nous nous intéresserons donc essentiellement à l'analyse syntaxique qui constitue une phase très importante de la compilation.

Dans le chapitre 0, nous rappelons brièvement quelques définitions concernant les langages formels puis nous décrivons succintement comment est définie la syntaxe d'Algol 68.

Dans le chapitre 1, nous montrons quels sont les problèmes que pose l'utilisation de cette syntaxe et nous proposons une méthode efficace pour les résoudre. Cette méthode est fondée sur le fait qu'en dépit de la très grande généralité des langages décrits par les grammaires du type de celle d'Algol 68, il est possible de trouver dans cette dernière certaines particularités et de les exploiter. Les résultats de ce chapitre peuvent être utilisés directement pour réaliser l'analyse syntaxique de programmes, néanmoins il nous a paru plus efficace de réaliser celle-ci en plusieurs phases et donc de dissocier la reconnaissance des structures, la recherche des définitions et la vérification de compatibilités.

Nous avons donc été conduits à déduire de la grammaire d'Algol 68 une c-grammaire pour effectuer la première phase de l'analyse. Ainsi au chapitre 2, nous proposons une méthode assez générale qui permette de trouver une c-grammaire décrivant un surensemble d'Algol 68. Cette méthode utilise d'ailleurs

certaines idées du chapitre 1.

En Algol 68, le nombre des différents genres de valeur n'est pas limité, il est décrit par un langage formel et le programmeur peut "inventer" autant de nouveaux genres qu'il le désire. Il doit donc préciser dans son programme les genres qu'il utilise. Ces genres eux-mêmes doivent respecter certaines règles qu'il est donc nécessaire de contrôler. Nous décrivons au chapitre 3 comment il est possible de représenter ces genres dans la machine qui fera la traduction.

Ensuite dans la première partie du chapitre 4, nous proposons divers algorithmes qui permettent de réaliser les contrôles mentionnés ci-dessus.

Dans un programme Algol 68, il est possible de préciser le même genre de diverses manières ; autrement dit la représentation des genres utilisée n'est pas injective. La représentation interne proposée au chapitre 3 se déduit directement de la précédente et elle n'est pas non plus injective. Dans la dernière partie du chapitre 4, nous proposons deux représentations injectives de tous les genres utilisables en Algol 68. (En corollaire nous montrons qu'il existe une relation d'ordre sur l'ensemble des genres).

Enfin la vérification de la compatibilité des genres implique la recherche des antécédents des opérateurs. En effet, il est possible en Algol 68, de définir soi-même le sens des opérateurs. Pour conserver certaines habitudes acquises le langage permet de donner plusieurs significations à un même symbole d'opérateur ; ainsi l'action représentée par un opérateur dépend du contexte dans lequel il est utilisé et plus particulièrement du genre des opérandes auquel s'applique cet opérateur. Par ailleurs, toujours pour conserver des habitudes acquises (ou pour en introduire de nouvelles), on définit de façon rigoureuse dans quelles conditions les genres sont compatibles ou non ; ainsi dans certains cas les genres sont fixés impérativement, mais dans d'autres une certaine liberté est laissée, c'est-à-dire que les genres peuvent être "modifiés". Ces libertés s'appliquent naturellement aux genres des opérandes. L'identification des opérateurs et le traitement des modifications (ou contrôle des compatibilités de genres) sont donc très liés. C'est l'objet du chapitre 5 que de proposer des solutions à tous ces problèmes (l'accent est mis sur l'identification des opérateurs).

Les études proposées sont préliminaires ; elles ne doivent pas être considérées comme définitives. D'autre part, bien qu'il semble plus efficace de séparer la compilation en plusieurs phases, celles-ci intéragissent énormément

les unes sur les autres et, comme les problèmes de représentation des valeurs n'ont pas été abordés, il est fort possible que certaines solutions proposées doivent être modifiées pour les adapter à un compilateur réel.

# CHAPITRE 0

## NOTATIONS, DEFINITIONS ET TERMINOLOGIE .

Nous noterons A-B la différence de deux ensembles A et B.

$$A-B = \{x \mid x \in A, x \notin B\}$$

Nous noterons card A le nombre d'éléments d'un ensemble fini A.

Par définition :

- un alphabet est un ensemble fini non vide ;
- un mot de longueur k ($k \in \mathbb{N}$) sur un alphabet A est une suite ordonnée $x_1, x_2, \ldots, x_k$ de k éléments de A ; une telle suite sera notée $x_1\ x_2 \ldots x_k = w$ et sa longueur k sera représentée par $|w|$ ;
- le mot vide est le mot de longueur nulle que nous noterons $\Lambda$ ;
- deux mots $x_1\ x_2 \ldots x_s$ et $y_1\ y_2 \ldots y_r$ sur A sont égaux si
  a) $r = s$
  b) $x_i = y_i$ pour $i = 1, 2, \ldots, r(=s)$.

Concaténation de deux mots :

Pour deux mots $x = x_1\ x_2 \ldots x_s$ et $y = y_1\ y_2 \ldots y_r$ sur A, c'est le mot, noté x.y (ou plus simplement xy), obtenu en faisant suivre le dernier élément de x par ceux de y : $x.y = x\ y = x_1\ x_2 \ldots x_s\ y_1\ y_2 \ldots y_r$.

Le monoïde libre engendré par un alphabet A est l'ensemble constitué de tous les mots sur A et du mot vide ; cet ensemble est noté $A^*$.

Le produit de deux parties Q et R d'un monoïde libre $A^*$ est l'ensemble noté QR et défini comme suit :

$$\forall\ Q, R \subset A^* : QR = \{qr \mid q \in Q, r \in R\}$$

Par ailleurs,

$\forall\ E \subset A^*$ nous noterons $E^0 = \{\Lambda\}$,

$$E^n = E^{n-1}\ E \text{ pour } n \geq 1$$

et $$E^* = \bigcup_{n=0}^{n\to\infty} E^n.$$

D'une manière analogue,

$\forall w \in A^*$ nous noterons $w^0 = \Lambda$,

$w^n = w^{n-1} w$ pour $n \geq 1$,

$w^* = \{w^n \mid n \in \mathbb{N}\}$

et $w^+ = \{w^n \mid n \in \mathbb{N}^+\}$

Un langage sur un alphabet A est une partie du monoïde libre engendré par A.

## C- LANGAGES ET C-GRAMMAIRES

Une c-grammaire est un quadruplet $G = (V, \sum, P, \sigma)$ où

- $\sum$ est un alphabet dit terminal dont les éléments sont appelés terminaux ;
- $V-\sum$ un alphabet dit non-terminal dont les éléments sont appelés non-terminaux ou encore variables ;
- P est une partie finie de $(V-\sum) \times V^*$ ; un élément $(\xi, v)$ de P est noté $\xi \to v$ et appelé production ;
- $\sigma$ un élément particulier de $V-\sum$ appelé axiome.

Une relation binaire notée $\underset{G}{\Longrightarrow}$ est définie entre deux éléments de $V^*$ comme suit :

$\forall x, y \in V^* : x \underset{G}{\Longrightarrow} y$ si

a) $\exists z_1, z_2, v \in V^*$,

b) $\exists \xi \in V-\sum$,

c) $\exists (\xi \to v) \in P$,

tels que $x = z_1 \xi z_2$ et $y = z_1 v z_2$

La cloture transitive de cette relation sera appelée dérivation et notée $x \overset{*}{\underset{G}{\Longrightarrow}} y$.

Nous noterons encore $L(x) = \{y \mid y \in \sum^*, x \overset{*}{\underset{G}{\Longrightarrow}} y\}$

$$L(G) = L(\sigma)$$

L(G) est le langage engendré par la grammaire G ; un c-langage est un langage engendré par une c-grammaire.

Une c-grammaire est réduite si

a) $\forall\ \xi \in V-\sum,\ \exists\ u, v \in V^*$ tels que $\sigma \overset{*}{\underset{G}{\Longrightarrow}} u\ \xi\ v$

b) $\forall\ \xi \in V-\sum,\ L(\xi) \neq \emptyset$

## DEFINITION DE LA SYNTAXE D'ALGOL 68

Métagrammaire :

C'est un triplet $G_M = (M, \sum, \Pi)$ où

- M est un alphabet dont les éléments sont appelés métanotions (elles sont représentées par des suites finies de signes syntaxiques majuscules),

- $\sum$ est un alphabet dont les éléments sont appelés signes syntaxiques minuscules $\sum = \{a, b, c, \ldots, y, z\}$ ;

- $\Pi$ est une partie finie de $M \times H^*$ avec $H = M \cup \sum$ ; les éléments de $H^+$ s'appellent des hypernotions ; un élément $(m, u)$ de $\Pi$ se note $m \to u$ et est appelé règle de production pour une métanotion.

Une relation binaire notée $\underset{G_M}{\Longrightarrow}$ est définie entre deux éléments de $H^*$ comme suit :

$\forall\ x, y \in H^*$ : $x \underset{G_M}{\Longrightarrow} y$ si

a) $\exists\ z_1, z_2, u \in H^*$,

b) $\exists\ m \in M$,

c) $\exists\ (m \to u) \in \Pi$,

tels que $x = z_1\ m\ z_2$ et $y = z_1\ u\ z_2$.

Nous noterons aussi $x \overset{*}{\underset{G_M}{\Longrightarrow}} y$ la cloture transitive de cette relation binaire.

D'une manière symbolique (voir [3]) nous noterons encore :

- $x \overset{\infty}{\underset{G_M}{\Longrightarrow}} y$, avec $y \in \Omega$ ensemble des suites non nécessairement finies de signes syntaxiques minuscules, si $\exists\ x = x_1 \underset{G_M}{\Longrightarrow} x_2 \underset{G_M}{\Longrightarrow} \ldots \underset{G_M}{\Longrightarrow} x_p \underset{G_M}{\Longrightarrow} \ldots$,

- $L(x) = \{y \mid y \in \Omega,\ x \overset{\infty}{\underset{G_M}{\Longrightarrow}} y\}$ l'ensemble des productions terminales de x.

Remarquons que $x \overset{*}{\underset{G_M}{\Longrightarrow}} y$ implique $x \overset{\infty}{\underset{G_M}{\Longrightarrow}} y$.

Pseudogrammaire associée à la métagrammaire

C'est un quintuplet $G_p = (M, \sum, \Delta, \Gamma, S)$ où

- M et $\sum$ ont été définis ci-dessus ;
- $\Delta$, ensemble des hypernotions droites, est une partie finie de $H^+$ ;
- $\Gamma$, ensemble des hypernotions gauches, est une partie finie de $H^+$ ;
- $S$, ensemble de schémas de règle, est une partie finie de $\Gamma \times \Delta^+$

Un élément de $S$ est noté :

$$\gamma : \delta_1, \delta_2, \ldots, \delta_r.$$

avec $\gamma \in \Gamma$ et $\delta_i \in \Delta$, $1 \leq i \leq r$.

Nous noterons de la même manière les éléments de $\Gamma \times \Delta^+$ que nous appellerons aussi, par abus de langage, des schémas de règle.

Remplacement compatible d'une métanotion dans un schéma de règle

Soit $m \in M$, w une production terminale de m et $s \in S$

$$s = x_1\ m\ x_2\ m \ldots x_{p-1}\ m\ x_p :$$
$$x_{p+1}\ m\ x_{p+2}\ m \ldots x_{q-1}\ m\ x_q,$$
$$\ldots, \ldots,$$
$$x_r\ m\ x_{r+1}\ m \ldots x_{t-1}\ m\ x_t.$$

avec $x_i \in (\sum \cup (M-\{m\}))^*$, $i = 1, 2, \ldots, t$

Par définition, le schéma de règle s' :

$$s' = x_1\ w\ x_2\ w \ldots x_{p-1}\ w\ x_p :$$
$$x_{p+1}\ w\ x_{p+2}\ w \ldots x_{q-1}\ w\ x_q,$$
$$\ldots, \ldots,$$
$$x_r\ w\ x_{r+1}\ w \ldots x_{t-1}\ w\ x_t.$$

est obtenu, à partir du schéma de règle s, par remplacement compatible de la métanotion m par sa production terminale w.

Nous noterons $s \underset{G_M}{\vdash} s'$ cette relation et $s \overset{*}{\underset{G_M}{\vdash}} s'$ sa cloture transitive.

Ces notations seront utilisées aussi pour la restriction de ces relations aux hypernotions constituantes des schémas de règle.

Grammaire d'Algol 68 associée à $G_M$ et $G_p$

C'est un quadruplet $G = (V, \mathcal{L}, P, \sigma)$ où

- $\mathcal{L}$ est une partie de $\sum^+ \{\text{symbol}\}$, ensemble des symboles ;
- $N = V - \sum$, ensemble des notions ; c'est une partie de $\Omega$ définie par

  $\xi \in N \subset \Omega \Leftrightarrow \exists\ \gamma \in \Gamma$ tel que $\gamma \overset{*}{\underset{G_M}{\vdash}} \xi$

- P est une partie de $N \times \Omega^*$ ; un élément de P s'appelle une règle de production pour une notion ; ces éléments seront notés

  $\xi : u_1, u_2, \ldots, u_k.$ avec $u_j \in \Omega$, $1 \leq j \leq k$ et ils sont définis par

  $p \in P$, $p = (\xi : u_1, u_2, \ldots, u_k.)$

  $\Leftrightarrow \exists\ s \in S$, $s = (\gamma : \delta_1, \delta_2, \ldots, \delta_k.)$

  tel que $s \overset{*}{\underset{G_M}{\vdash}} p$

- $\sigma$ est un élément particulier de $N$ que nous appellerons aussi axiome.

Le langage engendré par la grammaire G d'Algol 68 l'est de la même manière que dans le cas d'une c-grammaire. Ce langage s'appelle le langage strict par opposition au langage étendu ; ce dernier est obtenu à partir du premier à l'aide d'extensions, c'est-à-dire de simplifications d'écriture. Le langage de représentation est obtenu en supprimant les virgules (,) qui séparent les symboles et en remplaçant chacun de ceux-ci par l'une de leurs représentations.

Dans ce qui suit les références précédées de la lettre R renvoient à [1].

# CHAPITRE 1

## ALGORITHME POUR UNE UTILISATION EFFICACE DE LA GRAMMAIRE D'ALGOL 68.

La syntaxe du langage Algol 68 est décrite pour partie de façon formelle à l'aide d'une *grammaire* G (voir chapitre 0), et pour partie de façon informelle au moyen de phrases écrites en langue Anglaise. Cette deuxième partie concerne essentiellement les conditions de contexte (voir chapitre 4) et le processus d'identification (chapitre 5). Nous ne nous intéresserons dans les deux premiers chapitres qu'à la grammaire d'Algol 68. Celle-ci a la forme d'une c-grammaire, c'est-à-dire qu'elle est constituée d'un vocabulaire terminal, d'un vocabulaire non-terminal, d'un ensemble de règles de production et d'un élément privilégié du vocabulaire non-terminal, l'axiome. La grammaire d'Algol 68 diffère essentiellement d'une c-grammaire par le fait que le nombre de ses règles de production et celui des éléments de son vocabulaire non-terminal ne sont pas bornés, ce qui implique qu'on ne peut énoncer explicitement toutes ces règles et tous ces éléments. La définition formelle de la syntaxe du langage fournit donc un moyen d'engendrer des règles de production à l'aide de schéma de règles comme nous l'avons vu au chapitre 0.

Un mot du langage est engendré par un mécanisme classique de dérivation à partir de l'axiome de façon à obtenir une production terminale de cet axiome. Une dérivation de la forme :

$$u\ \xi\ v \underset{G}{\Longrightarrow} u\ w\ v \qquad \text{avec} \qquad u, w, v \in V^{*} \text{ et } \xi \in N$$

utilise une règle de production pour la notion $\xi$. Deux éventualités peuvent alors se présenter :

- ou bien il existe dans l'ensemble $S$ un schéma qui constitue en lui-même une règle de production pour la notion $\xi$,

. ou bien il existe un schéma de règle qui peut engendre une telle règle de production en remplaçant de manière compatible ses métanotions constituantes par des productions terminales de celles-ci.

Dans une règle de production pour une notion, il est possible que les protonotions droites ne soient pas des notions, c'est-à-dire qu'il n'existe pas de règle de production permettant de poursuivre la dérivation.

Par exemple : 'row of real base : row of real denotation.' est une règle de production pour 'row of real base', mais 'row of real denotation' n'est pas une notion car il n'existe aucune règle de production où cette protonotion figure en partie droite.

Etant donnée une protonotion, le premier objectif de l'algorithme est de *déterminer si c'est une notion* et dans l'affirmative de fournir le ou les *schémas* qui peuvent engendrer directement ou non une règle de production pour cette notion.

Exemples - étant donnée la protonotion 'row of boolean denotation' l'algorithme concluera que ce n'est pas une notion ;

- étant donnée la protonotion 'formal lower bound' l'algorithme fournira les schémas suivants :

formal LOWPER bound :
    strict LOWPER bound option, either symbol.
formal LOWPER bound :
    strict LOWPER bound option, flexible symbol option.

Dans le cas où il s'agit d'une notion et où il apparait des métanotions identiques dans l'hypernotion gauche et les hypernotions droites des schémas trouvés, il est nécessaire d'identifier, dans cette notion, les productions terminales des métanotions qui apparaissent dans l'hypernotion gauche des schémas, afin d'engendrer la règle de production correspondante. Nous appelerons cette reconnaissance la *filiation* entre la protonotion donnée et l'hypernotion gauche d'un schéma. Ce sera le deuxième objectif de l'algorithme.

Exemple : pour la notion 'formal lower bound' et les schémas associés de l'exemple précédent, il s'agit de reconnaitre que la protonotion 'lower' est une production terminale de la métanotion LOWPER. Ceci permettra d'engendrer, par remplacement compatible de LOWPER par 'lower' dans les schémas trouvés, les règles de production :

formal lower bound :
    strict lower bound option, either symbol.

formal lower bound :
strict lower bound option, flexible symbol option.

## 1 - TERMINOLOGIE

Précisons d'abord quelques concepts que nous utiliserons par la suite. Dans le rapport de définition du langage ([1]), les schémas qui ont la même hypernotion gauche sont regroupés par mise en facteur de celle-ci. Ainsi des schémas de la forme

(1) $\gamma : \delta_1, \delta_2, \ldots, \delta_r.$, $\gamma : \delta'_1, \delta'_2, \ldots, \delta'_s.$ et $\gamma : \delta''_1, \delta''_2, \ldots, \delta''_t.$

sont fournis sous la forme

(2) $\gamma : \delta_1, \delta_2, \ldots, \delta_r ; \delta'_1, \delta'_2, \ldots, \delta'_s ; \delta''_1, \delta''_2, \ldots, \delta''_t.$

Nous appellerons *règles de grammaire* de tels regroupements de schémas.

Les schémas tels que (1) engendrent, par remplacement compatible, des règles de production qui ont même notion gauche ; ces règles peuvent être regroupées de façon analogue par mise en facteur de la notion commune. Nous appellerons *règle syntaxique effective* le regroupement ainsi obtenu.

Nous appellerons *membre gauche* d'une règle de grammaire (resp. d'une règle syntaxique effective) l'hypernotion (resp. la notion) gauche commune aux schémas (resp. règles de production) regroupés.

Enfin nous ne considérerons que les règles de grammaire non précédées d'un "astérisque" et fournies dans le rapport de définition, aux chapitres 2 à 3 inclus, dans les paragraphes dont le titre commence par "Syntax". Les règles précédées d'un astérisque n'ont pas été prises en considération pour des raisons d'encombrement de la mémoire lors d'une application pratique ; celà ne nuit pas à la généralité de ce que nous dirons par la suite car ces règles n'ont été données que dans un but sémantique ; elles ne servent pas à la génération d'un programme, mais elles permettent de désigner des classes d'unités syntaxiques lors de la description de l'élaboration d'un programme. Il n'y aurait, de toute manière, aucun inconvénient à les ajouter à celles que nous avons retenues.

Avec la terminologie ainsi introduite, l'algorithme proposé

- permet, à partir d'une protonotion susceptible de constituer le membre gauche d'une règle syntaxique effective, de trouver la ou les règles de grammaire dont les membres gauches engendrent la protonotion donnée,

- donne la filiation entre ce ou ces membres gauches et la protonotion donnée.

## 2 - FORMALISATION DU PROBLEME

Dans ce qui suit, nous adopterons les notations du chapitre 0. L'ensemble des membres gauches de toutes les règles de grammaire est fini. L'ensemble d'hypernotions $\Gamma$ est donc fini ; en notant card $\Gamma$ le nombre de ses éléments, nous conviendrons de les indicer par un entier i, $1 \leq i \leq$ card $\Gamma$. Nous désignerons parfois les éléments de $\Gamma$ par la référence de la règle de grammaire correspondante dans le rapport.

Nous avons cherché à ramener ce problème à un problème simple sur les c-grammaires. Pour celà, notons $G = (V, \sum, P, \sigma)$ la c-grammaire définie par :

$V = M \cup \sum \cup \{\sigma\}$, $\sigma \notin M \cup \sum$,

et $P = \Pi \cup \{\sigma \rightarrow \gamma_i \mid \gamma_i \in \Gamma, 1 \leq i \leq \text{card } \Gamma\}$.

Le langage engendré par la c-grammaire $G$ est différent de $N$ ; en effet :

1) il existe un cas de remplacement compatible de métanotions dans le membre gauche d'une règle de grammaire (voir [1] règle R.8.2.5.1.b). Ce membre gauche est le suivant :

'strongly widened to structured with REAL field letter r letter e and REAL field letter i letter m FORM'

Cette hypernotion engendre uniquement les notions obtenues en y remplaçant les deux occurences de REAL par une même production terminale de cette métanotion et l'occurence de FORM par une production terminale de FORM. Or le mécanisme d'engendrement des mots de $L(G)$ fournira en plus, pour cette hypernotion, les protonotions obtenues en y remplaçant les deux occurences de REAL par deux productions terminales différentes de la métanotion REAL. Ce cas de remplacement compatible dans un membre gauche est le seul pour l'ensemble des membres gauches.

2) $L(G)$ est un sous-ensemble du monoïde libre engendré par $\sum$, il ne contient donc pas de mots de "longueur infinie" ; or les notions peuvent être, selon la définition de [1], des suites infinies de signes syntaxiques minuscules. Il en est ainsi pour les unités syntaxiques dont le mode est un mode récursif (voir chapitre 3) autorisé par les conditions de contexte ; par exemple dans la déclaration *mode a = struct (rep a x), struct (rep a x)* est un 'actual

**structured with reference to structured with reference to structured ... x letter x letter x declarer'.** Sur le plan pratique les protonotions de longueur infinie ne peuvent être manipulées directement. Selon une étude axiomatique de la nature des protonotions que l'on trouvera dans [3], l'ensemble des protonotions forme un groupoïde sur $\sum$. Cette étude montre qu'il est possible d'identifier la partie de $N$ constituée de protonotions de longueur finie à l'ensemble des mots non vides sur $\sum$. Dans la suite nous nous limiterons au cas fini.

Il est clair que $L(G)$ est un sur-ensemble de la partie de $N$ définie ci-dessus.

Le problème que nous avons énoncé peut donc, après cette restriction, se formuler de la manière suivante :

étant donné $w \in \sum \sum^*$ (w est donc une protonotion non vide de longueur finie)

(a) w appartient-il ou non à $L(G)$ ?

(b) si $w \in L(G)$,

trouver tous les $\gamma_i \in \Gamma$ tels que $\sigma \underset{G}{\Rightarrow} \gamma_i \overset{*}{\underset{G}{\Rightarrow}} w$

(c) si $w \in L(G)$, pour chacun des $\gamma_i$ trouvé en (b)

et pour tout $m \in M$ et telle que $\gamma_i = u\ m\ v$, $u, v \in V^*$

trouver $z \in V^*$ tel que i) $w = x\ z\ y$

ii) $m \overset{*}{\underset{G}{\Rightarrow}} z$

(d) si le membre gauche trouvé en (b) est celui cité dans la remarque 1) ci-dessus, vérifier que les sous-mots de w constituant les productions terminales de **REAL** trouvés en (c) sont les mêmes.

Remarque 1 :

Pour un même $\gamma_i$ la filiation obtenue en (c) peut ne pas être unique,

Exemple : $\gamma_i$ = one out of LMOODSETY MOOD RMOODSETY mode FORM (R.8.2.4.1.b)

w = one out of boolean and integral and real mode base

les trois filiations suivantes sont possibles :

ou bien LMOODSETY $\overset{*}{\underset{G}{\Longrightarrow}}$ boolean,

MOOD $\overset{*}{\underset{G}{\Longrightarrow}}$ integral,

et RMOODSETY $\overset{*}{\underset{G}{\Longrightarrow}}$ real,

ou bien LMOODSETY $\overset{*}{\underset{G}{\Longrightarrow}}$ boolean and integral,

MOOD $\overset{*}{\underset{G}{\Longrightarrow}}$ real,

et RMOODSETY $\overset{*}{\underset{G}{\Longrightarrow}}$ $\Lambda$,

ou encore LMOODSETY $\overset{*}{\underset{G}{\Longrightarrow}}$ $\Lambda$,

MOOD $\overset{*}{\underset{G}{\Longrightarrow}}$ boolean,

et RMOODSETY $\overset{*}{\underset{G}{\Longrightarrow}}$ integral and real

Il y a un deuxième membre gauche susceptible de fournir plusieurs filiations ; il s'agit de l'hypernotion REFETY ROWSETY ROWWSETY NONROW slice.
Ces deux cas sont les seuls. Suivant la méthode utilisée il sera possible d'obtenir toutes les filiations possibles, ou bien une seule choisie de façon arbitraire.

Remarque 2 :

En raison du regroupement des schémas de règle que nous avons décrit précédemment, il n'existe le plus souvent, pour une notion donnée, qu'un seul membre gauche vérifiant la condition énoncée en (b) ; cependant dans le cas des notions engendrées par les membres gauches des règles de grammaire de références R.7.1.1.ee, et R.7.1.1.ff, il est possible de trouver plusieurs membres gauches qui sont :

LOSETY closed LMOODSETY LMOOD end BOX
et LOSETY closed LMOODSETY LMOOD end LMOOT BOX.

Les ensembles de notions qu'ils engendrent ont une intersection non vide,

Cette remarque et la précédente montrent que la grammaire G est ambigue.

Remarque 3 :

La vérification décrite en (d) constitue un cas très particulier qui ne pose aucun problème. Il n'en sera plus question par la suite.

## 3 - PREMIERE METHODE

Un algorithme d'analyse syntaxique multiple utilisant la c-grammaire $G$ permet de résoudre à la fois (a), (b) et (c).

En effet, un tel algorithme répond à la question posée en (a) et, si $w \in L(G)$, il construit tous les arbres de génération des dérivations du mot w dans la grammaire $G$. Pour chaque arbre ainsi obtenu, les étiquettes des extensions du sommet d'étiquette $\sigma$ permettent, en les concaténant de gauche à droite, de reconstituer un membre gauche $\gamma_i$ tel que $\sigma \overset{*}{\underset{G}{\Rightarrow}} \gamma_i$ ; (b) est ainsi résolu.

Enfin les étiquettes des feuilles de chaque sous-arbre dont le sommet
- est une extension du sommet d'étiquette $\sigma$,
- a pour étiquette une métanotion m,

permettent de la même façon, c'est-à-dire en les concaténant de gauche à droite, de reconstituer les sous-mots se dérivant à partir de m, donc de résoudre (c).

De tels algorithmes sont classiques [4]. Il suffit d'en choisir un qui ne nécessite pas la transformation d'éléments de $\Gamma$ pour sa mise en oeuvre. Remarquons que cette solution fournit toutes les filiations possibles puisqu'il s'agit d'un algorithme d'analyse multiple.

Cette solution paraît simple dans son principe ; il nous a cependant semblé qu'elle serait peu efficace dans son utilisation pratique, en particulier en raison du très grand nombre de comparaisons qu'exige le codage signe syntaxique par signe syntaxique des protonotions. De plus, seule la structure de l'arbre au voisinage de la racine (sommet d'étiquette $\sigma$) est intéressante ; cette méthode fournit donc des informations inutiles dans le cadre du problème posé. C'est pourquoi, nous avons été conduit à rechercher une autre méthode.

## 4 - DEUXIEME METHODE

Pour remédier au premier inconvénient mentionné dans le paragraphe précédent nous allons modifier la définition des protonotions, puis chercher à utiliser un simple algorithme de reconnaissance à la place d'un algorithme d'analyse multiple.

Les protonotions sont des suites de signes syntaxiques minuscules. Il se trouve que les notions dont dépend l'axiome de la grammaire G d'Algol 68 se décomposent sur un ensemble assez réduit de protonotions que nous appellerons

*protonotions simples*. Intuitivement, ces éléments sont obtenus en considérant les espaces introduits dans les règles de grammaire afin de rendre certaines notions plus lisibles si on les considère comme des parties de phrases écrites en Anglais.

Nous remplacerons donc la c-grammaire $G$ par une nouvelle c-grammaire

$$G' = (V', \sum', P', \sigma) \text{ telle que :}$$

$\sum'$ soit l'ensemble des protonotions simples, c'est-à-dire des protonotions qui apparaissent sous forme de mots anglais

- dans les membres gauches des règles de grammaire,
- dans la métagrammaire et
- dans les symboles susceptibles d'être contenus dans une notion ;

nous avons ajouté à tous ces mots le suffixe 'ly' (à cause des règles de grammaire concernant les modifications) ;
nous avons aussi pris soin de détacher le 's' qui termine certaines protonotions simples (à cause de la règle de construction générale R.3.0.1.c).

On trouvera l'ensemble de ces protonotions simples dans un tableau figurant en annexe ; les protonotions simples qui y figurent sont par ailleurs tronquées à dix signes syntaxiques, ce qui n'introduit pas d'ambiguités.

Une protonotion sera alors une séquence non vide de protonotions simples séparées par des espaces (dans ce qui suit, chaque protonotion simple sera considérée, non pas comme une suite de signes syntaxiques minuscules, mais comme une entité non composite représentée par une telle suite). Cependant, une notion demeure un cas particulier de protonotion (voir chapitre 0).

$V' = M \cup \sum' \cup \{\sigma\}$

$P'$ soit obtenu à partir de l'ensemble $P$ en mettant en évidence la décomposition en protonotions simples des protonotions qui apparaissent dans les éléments de $P$. Il suffit, pour réaliser cette transformation, de considérer les espaces, introduits par les auteurs de [1] afin de rendre plus lisibles les hypernotions, comme des occurences de l'opérateur de concaténation entre protonotions simples ou métanotions. Ainsi les règles de production de $P$ pour la métanotion PRAM se représentent par :

PRAM → procedurewithLMODEparameterandRMODEparameterMOID
PRAM → procedurewithRMODEparameterMOID,

en pratique elles sont fournies sous la forme :

PRAM → procedure with LMODE parameter and RMODE parameter MOID
PRAM → procedure with RMODE parameter MOID,

qui représentent les règles de production de $P'$ pour la métanotion PRAM.

Remarquons tout d'abord que $\sum'$ contient $\sum$ puisque les signes syntaxiques minuscules apparaissent comme protonotions simples dans la métagrammaire (productions terminales de la métanotion ALPHA). La décomposition de certaines protonotions (en particulier des notions) peut donc ne pas être unique ; nous adopterons la décomposition la plus naturelle, c'est-à-dire celle qui n'utilise les signes syntaxiques minuscules comme protonotions simples qu'à la suite de la protonotion **'letter'**. Les notions susceptibles d'être engendrées par des membres gauches contenant la métanotion NOTION peuvent aussi se décomposer sur $\sum'$ mais certaines peuvent ne pas se décomposer sur $\sum' - \sum$.

Par exemple : **'heapsymboloption'** peut se décomposer
en **heap symbol option**
ou en **h e a p  s y m b o l  option**

Par ailleurs **'trucoption'** est une notion mais ce n'est pas une notion utile car elle ne dépend pas de l'axiome de la grammaire d'Algol 68 ; en effet aucun schéma de règle ne peut engendrer de règle de production comportant en partie droite une telle notion. Cette notion peut se décomposer sur $\sum'$ mais non sur $\sum' - \sum$. Nous avons laissé à l'utilisateur la possibilité de décomposer soit sur $\sum' - \sum$ soit sur $\sum$ les protonotions se dérivant de NOTION dans les notions obtenues à partir des schémas contenant cette métanotion. Nous verrons ci-dessous comment ceci est réalisé en pratique.

Au lieu d'utiliser un algorithme d'analyse syntaxique multiple, nous avons cherché à nous limiter à un simple algorithme de reconnaissance. Nous avons choisi l'algorithme dit "des suites possibles" [5]. Un tel algorithme ne conserve pas la structure de l'arbre de génération et il ne permet que la résolution de la partie (a) du problème posé. Afin de résoudre en même temps la partie (b) nous avons procédé de la manière suivante :

L'algorithme des suites possibles est normalement initialisé en plaçant dans la pile principale l'axiome de la c-grammaire ; ici, nous plaçons successivement chaque élément $\gamma_i$ de $\Gamma$ dans la pile principale, puis nous lançons l'algorithme de reconnaissance en mémorisant l'indice i ; chaque fois que la

protonotion w est reconnue appartenir au language engendré par la c-grammaire $G'$, l'indice i fournit immédiatement l'un des membres gauches cherchés. De plus, chaque fois que la métanotion NOTION se trouve au sommet de la pile principale (elle doit donc être dérivée), elle est remplacée

- soit par la protonotion simple de w analysée à ce moment,
- soit par cette même protonotion simple suivie de la métanotion NOTION.

Cette solution au problème posé par l'existence de cette métanotion, insuffisante au point de vue théorique est en pratique satisfaisante. Cette manière de procéder a par ailleurs l'avantage de ne pas augmenter le nombre de production de $P'$ en ne créant certaines règles de productions que lorsqu'elles sont nécessaires.

Pour obtenir la filiation souhaitée dans la troisième partie du problème posé, l'algorithme est modifié de la manière suivante :

tous les éléments constituant un membre gauche, c'est-à-dire soit des protonotions simples soit des métanotions, sont marqués lors du transfert de ce membre gauche dans la pile principale,

puis, lors du déroulement de l'algorithme de reconnaissance, la valeur du pointeur dans la protonotion w est mémorisée, si elle ne l'était déjà, à condition que la protonotion simple qui se trouve au sommet de la pile principale soit marquée et égale à la protonotion simple en cours d'analyse dans w. La valeur de ce pointeur est également mémorisée lors du rangement d'une "suite possible" en pile auxiliaire après les dérivations successives d'une métanotion marquée.

Le nombre des métanotions figurant dans un même membre gauche étant assez réduit en général, les informations que constituent l'ensemble des valeurs ainsi mémorisées est suffisant pour obtenir la filiation..

Dans le cas où le $\gamma_i$ n'est constitué que de protonotions simples, l'ensemble de ces valeurs représente la suite croissante des entiers positifs inférieurs ou égaux à la longueur de w ; il n'y a dans ce cas aucun sous-mot à isoler.

Exemple : $\gamma_i$ = structured with row of character field letter aleph digit one transformat

Dans le cas où le $\gamma_i$ ne comporte qu'une seule métanotion il est clair que les valeurs ainsi mémorisées sont les indices dans w des protonotions simples encadrant le sous-mot qui se dérive de cette métanotion et celui de la première

protonotion simple de ce sous-mot, il est alors aisé d'isoler ce sous-mot.

Exemple : $\gamma_i$ = formal MODE parameter

w = formal structured with row of character field letter aleph

1 2 3 4 5 6 7 8 9

digit one parameter

10 11 12

(les indices mémorisés sont 1, 2 et 12).

De même dans d'autres cas simples, où il peut y avoir plusieurs métanotions, la reconnaissance des sous-mots à l'aide de ces indices est tout aussi facile.

Il se pose un problème lorsqu'une métanotion marquée se trouve dérivée plusieurs fois (il y a plusieurs suites possibles contenant cette métanotion), car dans ce cas certains des indices retenus ne doivent pas être utilisés. Cette situation se produit en particulier lorsqu'une métanotion à règles de production récursives et suivie d'une autre métanotion dans un membre gauche. Fort heureusement cette situation ne se produit qu'une seule fois dans un même membre gauche ; il suffit donc de connaître, pour de tels membres gauches (qui sont en nombre assez réduit), le nombre d'indices à conserver à droite et le nombre à conserver à gauche pour obtenir la filiation.

Exemple : $\gamma_i$ = VICTAL ROWS NONSTOWED declarator

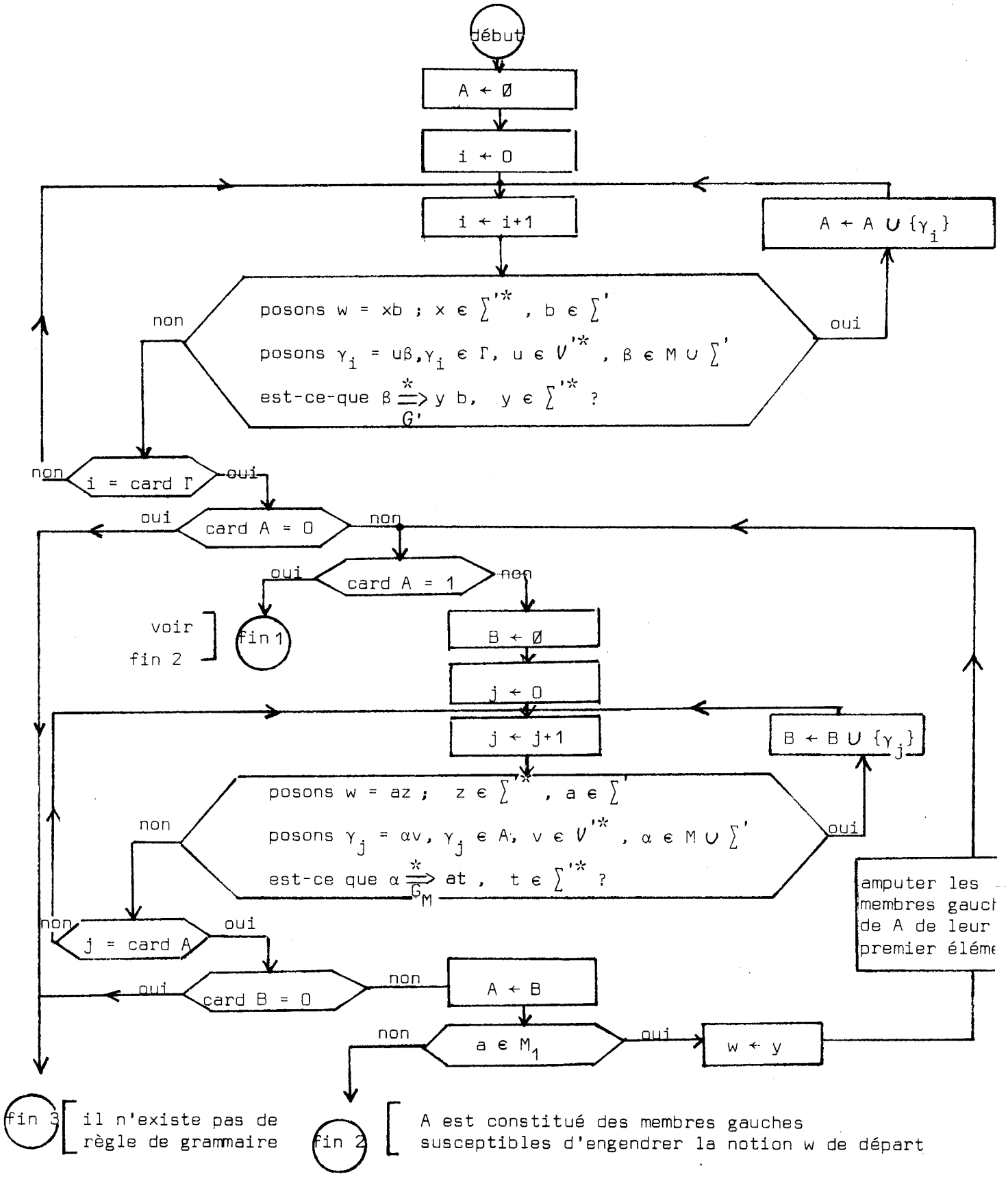
w = formal row of row of procedure real declarator

1 2 3 4 5 6 7 8

(les indices mémorisés sont 1, 2, 4, 6, 8),

il suffit d'en conserver, pour ce $\gamma_i$, deux à gauche et deux à droite pour obtenir l'ensemble d'indices corrects : 1, 2, 6, 8.

Il faut noter qu'en général l'algorithme des suites possibles exige quelques transformations de la grammaire mise en jeu (élimination des récursivités à gauche par exemple). Dans le cas présent, ces transformations n'entraînent pas de modifications des membres gauches $\gamma_i$. D'autre part, la méthode décrite pour obtenir la filiation permet de ne fournir qu'une seule filiation dans le cas où il y en a plusieurs (voir la remarque 1 du paragraphe 2) ; les membres gauches en question comportent d'ailleurs une métanotion récursive suivie d'une autre métanotion.




Organigramme de principe de l'algorithme d'amélioration de la deuxième méthode.

## 5 - AMELIORATION DE LA METHODE PRECEDENTE

Nous avons cherché à réduire, dans la méthode précédente, le nombre des membres gauches $\gamma_i$ à placer successivement en pile principale ; nous avons alors constaté que le procédé utilisé pour effectuer cette réduction était efficace et qu'il pouvait pratiquement suffire pour résoudre partiellement la recherche des membres gauches, par exemple dans le cas d'applications pédagogiques. Ce procédé est aussi intéressant dans le cas où les protonotions susceptibles de constituer le membre gauche d'une règle syntaxique effective sont fabriquées automatiquement à partir des hypernotions figurant dans les règles de grammaire à droite du deux-points ; en effet les cas où des protonotions engendrées ainsi ne sont pas des notions sont assez rares. De plus, il permet de se limiter à la connaissance de la dernière et des toutes premières protonotions simples de la notion considérée, ce qui est partiellement utile lorsque cette dernière est de longueur infinie.

Dans ce qui suit, nous supposerons toujours que les protonotions sont des séquences non vides de protonotions simples.

Le principe de la méthode consiste à éliminer, dans les membres gauches $\gamma_i$, ceux qui, à coup sûr, ne peuvent engendrer la protonotion donnée. Cette élimination est fondée sur la remarque suivante : les notions ont une signification en Anglais ; c'est pourquoi, elles sont toujours constituées d'une protonotion simple jouant le rôle "d'un substantif", éventuellement précédée d'une suite de protonotions simples jouant le rôle "d'adjectifs". La valeur sémantique attachée au "substantif" est très grande ; celà a pour conséquence que la dernière protonotion simple d'une notion caractérise en grande partie cette notion. La valeur sémantique des "adjectifs" qui précède ce "substantif" est inférieure à celle du "substantif" et est d'autant plus faible que l'adjectif est placé plus à droite dans la notion ; de façon plus précise, ce sont les premiers adjectifs qui sont les plus intéressants.

La méthode se décompose donc en deux phases :

Première phase :

- sélection, parmi tous les membres gauches, de ceux qui se *terminent*
  - . soit par la *dernière* protonotion de w,
  - . soit par une métanotion ayant parmi ses productions terminales au

moins une protonotion dont la *dernière* protonotion simple de w soit *facteur droit* (une telle métanotion sera appelée un antécédant de la protonotion simple).

(Il y a peu de membres gauches qui se terminent par une métanotion et par ailleurs le nombre de ces antécédents est assez réduit).

- les tests suivants sont ensuite réalisés :
  - . si aucun membre gauche n'a été sélectionné, c'est qu'il n'existe pas de règle syntaxique effective (w n'est donc pas une notion) ;
  - . si un seul membre gauche a été sélectionné, alors la solution de la partie (a) du problème initialement posé fournit immédiatement celle de la partie (b) ;
  - . si plusieurs membres gauches ont été sélectionnés, la seconde phase est effectuée.

Seconde phase :

- sélection, parmi les membres gauches issus de la première (ou de la deuxième) phase, de ceux qui *commencent*
  - . soit par la *première* protonotion simple de w,
  - . soit par une métanotion ayant parmi ses productions terminales au moins une protonotion dont la *première* protonotion simple de w soit *facteur gauche* (une telle métanotion sera aussi appelée un antécédent de la protonotion simple).
- même tests que dans la première phase.
  Cependant, si plusieurs membres gauches ont été sélectionnés à l'issue de cette seconde phase et si, de plus, la première protonotion simple de w peut être facteur gauche d'une production terminale d'une métanotion dont certaines productions terminales sont de longueur supérieure ou égale à deux (en pratique il s'agit des métanotions associées au modes),

  alors la réduction du nombre de membres gauches est arrêtée ; la recherche peut être poursuivie, parmi les membres gauches issus de la deuxième phase, à "la main", ou bien à l'aide de la deuxième méthode ;

  sinon, la deuxième phase est réitérée en amputant w de sa première protonotion simple et les membres gauches sélectionnés des antécédents de cette protonotion simple, ou de cette protonotion simple elle-même.

Pour mettre en oeuvre cette méthode qui permet de réduire le nombre de membres gauches susceptibles d'engendrer la protonotion w, il est nécessaire de dédoubler les membres gauches des règles de grammaire qui commencent par une métanotion ayant le vide parmi ses productions terminales (le dédoublement est aussi réalisé pour d'autres membres gauches où figurent de telles métanotions). En effet, pour de tels membres gauches il faudrait, dans la deuxième phase, rechercher aussi si le deuxième élément du membre gauche est la même protonotion simple que la première de w ou un antécédent de la première protonotion simple de w. Si le deuxième élément du membre gauche est de nouveau une métanotion ayant le vide parmi ses productions terminales, il faut itérer ce processus. Il nous a paru plus simple de décomposer les règles de grammaire commençant par ces membres gauches ; un tel membre gauche est remplacé par deux nouveaux membres gauches : l'un amputé de la métanotion de tête, et l'autre avec une production directe non vide de cette même métanotion. Dans le cas de la méta-grammaire d'Algol 68 cette production directe est toujours unique, ce qui justifie l'emploi du terme de dédoublement des membres gauches.

Exemple : SORTETY serial CLAUSE est remplacé par
serial CLAUSE et SORT serial CLAUSE

L'amélioration apportée par l'algorithme décrit ci-dessus permet essentiellement d'éviter de multiplier les analyses d'un mode, c'est-à-dire d'une production terminale de la métanotion MODE, alors que la dernière protonotion simple permet de différencier le membre gauche recherché par les éléments de Γ. Ceci est vrai aussi bien dans le cas où les notions commencent par un mode

(Exemples : MODE confrontation, MODE source, MOID cast, MODE cohesion, MODE base, etc) que dans celui où elles comportent un mode

(Exemples : SORTETY serial CLAUSE, SORTETY closed CLAUSE, SORTETY conditionnal CLAUSE, SORTETY unitary MOID clause, SORTETY MOID tertiary etc).

Les principes de l'algorithme décrits dans ce chapitre restent valables dans le cas d'une recherche des hypernotions droites susceptibles d'engendrer une notion donnée.

# CHAPITRE 2

## DEFINITION D'UN C-LANGAGE, SURENSEMBLE D'ALGOL 68.

Nous avons déjà vu aux chapitres précédents qu'une partie de la syntaxe d'Algol 68 était décrite par une grammaire G de forme "context-free". Lors de la génération d'un mot du langage à l'aide de cette grammaire, un nombre fini de règles de production est utilisé. Il est donc possible, en principe, d'associer à chaque programme, c'est-à-dire à chaque production terminale de l'axiome, une c-grammaire qui permette de réaliser une analyse syntaxique de ce programme (la grammaire G étant constituée de la réunion de toutes ces c-grammaires particulières) ; la construction de chacune de ces c-grammaires pourrait être réalisée, lors de l'analyse elle-même au fur et à mesure des besoins, par exemple en utilisant l'algorithme décrit au chapitre précédent.

L'objet du présent chapitre est de définir un procédé permettant d'obtenir directement à partir de la métagrammaire et la pseudogrammaire une c-grammaire qui engendre un c-langage contenant celui engendré par la grammaire G. L'intérêt essentiel réside dans le fait qu'il sera alors possible de réaliser une analyse de la structure "context-free" à l'aide d'une c-grammaire unique et construite une fois pour toute. Par ailleurs pour profiter au maximum de la puissance de la grammaire G, il est nécessaire de réaliser auparavant l'identification des identificateurs et indicateurs, ce qui complique l'analyse.

Le principe d'obtention de la c-grammaire repose sur le chaînage des schémas de règle de la pseudogrammaire (ce chaînage est matérialisé par les renvois qui suivent les hypernotions constituantes des schémas de règle) ; la structure obtenue à l'issue d'une analyse réalisée à l'aide de cette c-grammaire peut être affinée si certains schémas ont été décomposés par remplacement des métanotions, ou bien au contraire rendue plus grossière si certains schémas sont regroupés au moyen d'une seule règle de production.

Ainsi nous construirons d'abord une pseudogrammaire équivalente à celle fournie par [1], c'est-à-dire une pseudogrammaire permettant d'engendrer les règles de production de la grammaire G ; puis nous modifierons la pseudogrammaire obtenue en gardant sa puissance de description ; enfin nous déduirons la c-grammaire cherchée. Nous verrons aussi, à la fin de ce chapitre, dans quelle mesure

il est possible d'ajouter à cette c-grammaire des règles de production qui décrivent les extensions.

## 1 - TRANSFORMATIONS CONSERVANT LES PROPRIETES DE LA PSEUDOGRAMMAIRE

Elles consistent essentiellement à supprimer de certains schémas de règle un certain nombre de métanotions. Nous décrirons dans ce qui suit des transformations très générales, mais en pratique nous nous limiterons, parmi toutes ces transformations possibles, à un petit nombre d'entre elles.

### 1.1. Remplacement compatible de certaines métanotions par leurs productions terminales

Parmi les métanotions qui apparaissent dans des schémas de règle, il y en a qui engendrent un ensemble fini de productions terminales de longueur finie par dérivation dans la métagrammaire $G_M$. Dans une première étape nous allons supprimer des schémas de règles ces métanotions sans changer la grammaire G d'Algol 68.

Pour celà, nous créons de nouveaux schémas obtenus en remplaçant, de façon compatible, chacune de ces métanotions dans chacun des schémas où elle apparait, par chacune de ses productions terminales ; ceci implique la création d'autant de nouveaux schémas que cette métanotion a de productions terminales. Dans le cas où un même schéma contient plusieurs de ces métanotions, la transformation ainsi décrite sera appliquée successivement à chacune d'elle. Les nouveaux schémas obtenus remplacent les schémas initiaux.

Par exemple, un schéma de règle de la forme :

$$x_1\ m_1\ x_2\ m_2\ x_3 : x_4, x_5\ m_1\ x_6, x_7, x_8\ m_2\ x_1, x_{10}, x_{11}\ m_1\ x_{12}\ x_{13}, \ldots,$$
$$x_p\ m_1\ x_{p+1}, x_{p+2}$$

où $x_1, \ldots, x_{p+2}$ sont des protonotions, sera remplacé par les schémas

$$x_1\ y_1\ x_2\ y_2\ x_3 : x_4, x_5\ y_1\ x_6, x_7, x_8\ y_2\ x_9, x_{10}, x_{11}\ y_1\ x_{12}\ x_{12}, \ldots,$$
$$x_p\ y_1\ x_{p+1}, x_{p+2}$$

$\forall\ y_1 \in L(m_1)$, $\forall\ y_2 \in L(m_2)$, avec $L(m_1)$ et $L(m_2)$ bornés.

Le nombre des nouveaux schémas ainsi obtenus est borné, puisque nous avons supposé que le nombre de productions terminales des métanotions faisant l'objet de l'élimination était lui aussi borné. D'autre part il est clair que la grammaire G n'est pas modifiée ; en effet la transformation revient à réaliser en partie les étapes de construction des règles de production pour une notion (voir R.1.1.5.a), pour lesquelles il n'est imposé aucun ordre sur le choix des métanotions qui font l'objet d'un remplacement ; il est donc possible de commencer par celles qui engendrent un ensemble fini de productions terminales.

Pour cette première transformation, seules, parmi de telles métanotions, seront prises en considération les suivantes :

a) EMPTY, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE

Ces métanotions n'ont chacune qu'une seule production terminale ; elles ne servent donc que d'abréviations. Le remplacement des dernières permettrait aussi, si elle ne dépendait de l'identification des indicateurs adiques, de conserver la structure des priorités des formules.

b) LIST, SEPARATOR, PACK, ANY, VICTAL, VIRACT

Ces métanotions ont chacune plusieurs productions terminales. Les deux premières sont remplacées car leurs productions terminales entrainent l'utilisation de symbole différents. Le remplacement des deux dernières permettra d'obtenir une syntaxe "context-free" des déclareurs assez précise (analogue à celle que l'on peut trouver dans [6]).

c) THELSE, PRIMITIVE, LETTER, DIGIT

Ces métanotions ont aussi chacune plusieurs productions terminales ; leur remplacement par ces dernières est nécessaire car elles apparaissent dans des hypernotions susceptibles d'engendrer des symboles, c'est-à-dire des terminaux de la grammaire G (voir R.6.4.1.e, R.7.1.1.c, R.3.0.2.b et R.3.0.3.d).

d) COERCEND, FORM et PRIORITY

Ces remplacements permettent de transcrire dans la c-grammaire la structure résultant de la priorité des formules entre elles (voir cependant a) et avec les tertiaires, secondaires et primaires.

Les transformations ainsi décrites seront réalisées dans tous les schémas.

1.2. Remplacement compatible de certaines métanotions par certaines de leurs productions non terminales

Nous allons transformer la pseudogrammaire obtenue à la suite des transformations précédentes, en remplaçant cette fois certaines métanotions (à savoir celles dont les productions terminales sont des modes au sens de [1]) par des hypernotions qui ne sont pas nécessairement des productions terminales, mais de manière à ce que la réunion des ensembles de productions terminales de ces hypernotions soit le même ensemble que celui des productions terminales de la métanotion objet du remplacement.

De même cette transformation de la pseudogrammaire ne modifie pas la grammaire G. En effet, une *production terminale d'une métanotion* est obtenue par un processus de *remplacements successifs* de métanotions par leurs productions directes ; une *règle de production* pour une notion est obtenue par *remplacement de métanotions* par leurs productions *terminales* ; la transformation envisagée revient donc à dériver à l'aide d'une métagrammaire sensiblement modifiée, les métanotions dans les schémas de règles ; par ailleurs toute règle de production obtenue à l'aide du schéma initial peut être obtenue à l'aide de l'un des nouveaux schémas, puisque la réunion des productions terminales des hypernotions remplaçant la métanotion est égale à l'ensemble des productions terminales de la métanotion initiale.

Nous nous limiterons aux remplacements décrits ci-dessous, car nous avons pensé qu'ils permettraient d'obtenir à l'issue des différentes phases de l'algorithme, une c-grammaire suffisamment proche de la grammaire G en ce qui concerne les déclareurs de mode (voir aussi la remarque au paragraphe 1.1.a de ce chapitre). D'autre part les transformations seront appliquées à certains schémas et non à l'ensemble des schémas. En règle générale il s'agit des schémas de R.7 qui concernent la syntaxe des déclareurs et une seule métanotion par schéma sera remplacée.

a) MOID est remplacé par MODE et void dans les schémas R.7.1.1.a et R.3.3.4.1.a.

b) INTREAL est remplacé par INTEGRAL et REAL dans les schémas obtenus à partir de R.7.1.1.d.

c) MODE est remplacé par integral, real, boolean, character, format, long INTEGRAL, long REAL, structured with FIELDS, reference to STOWED,

reference to NONSTOWED, ROWS structured with FIELDS, ROWS NONSTOWED,PROCEDURE et UNITED,
dans les schémas obtenus à la suite des transformations précédentes à partir de R.7.1.1.b.

d) STOWED est remplacé par ROWS structured with FIELDS, ROWS NONSTOWED et structured with FIELDS dans les schémas déduits de R.7.1.1.h et R.7.1.1.m.

e) NONSTOWED est remplacé par integral, real, boolean, character, format, long INTEGRAL, long REAL, reference to STOWED, reference to NONSTOWED, PROCEDURE et UNITED dans les schémas obtenus à partir de R.7.1.1.k, R.7.1.1.n et R.7.1.1.p.

f) MOOD est remplacé par integral, real, boolean, character, format, long INTEGRAL, long REAL, reference to STOWED, reference to NONSTOWED, structured with FIELDS, ROWS structured with FIELDS, ROWS NONSTOWED et PROCEDURE dans le schéma R.7.1.1.jj.

## - TRANSFORMATIONS MODIFIANT LA GRAMMAIRE G

### 2.1. Traitement des constructions générales

Les schémas de règle qui contiennent la métanotion NOTION (schémas R.3.0.1 b,c,d,g,h,i) permettent d'engendrer une infinité de règles de production du langage strict. Parmi toutes ces règles, seules sont utilisables celles qui comportent à droite du deux-points des notions, c'est-à-dire des hypernotions qui peuvent être obtenues à partir des hypernotions figurant en partie gauche des schémas de la pseudogrammaire. Autrement dit la grammaire G n'est pas *réduite* car certaines notions ne dépendent pas de l'axiome de cette grammaire. La suppression de ces notions et des règles de production associées ne modifie pas le langage engendré par la grammaire G ainsi modifiée.

Nous supprimerons ces notions en substituant aux schémas qui décrivent des constructions générales de nouveaux schémas qui n'engendrent pas de telles règles de productions inutiles ; ces nouveaux schémas sont obtenus en remplaçant de façon compatible NOTION par des hypernotions déduites de la pseudogrammaire comme suit :

- toute hypernotion figurant à droite du deux-points d'un schéma, ne contenant pas la métanotion NOTION et se terminant par option (resp. list, sequence,

list proper, sequence proper, pack et package) fournit, par suppression de ce suffixe, une nouvelle hypernotion destinée à remplacer NOTION dans les schémas obtenus à partir de R.3.0.1.b (resp. R.3.0.1.d, R.3.0.1.g, R.3.0.1.h, R.3.0.1.i),

- toute hypernotion figurant à droite du deux-points d'un schéma, ne contenant pas la métanotion NOTION, commençant par chain of et se terminant par separated by list separators (resp. separated by sequence separators fournit, par suppression à la fois de ce préfixe et de ce suffixe, une nouvelle hypernotion destinée à remplacer NOTION dans l'un des schémas obtenu à partir de R.3.0.1.c par remplacement de SEPARATOR par list separator (resp. par sequence separator).

La création de ces nouveaux schémas entraine, dans certains cas, l'apparition de nouvelles hypernotions susceptibles de se terminer par les suffixes cités ci-dessus ; le processus est alors itéré, de manière à créer les nouveaux schémas correspondants. Naturellement un schéma qui existe déjà n'est pas recréé de nouveau (c'est le cas pour les schémas déduits de R.3.0.1.c). Les suffixes et préfixes cités étant en nombre fini dans la pseudogrammaire et le processus décrit réduisant, à chaque étape, le nombre de ceux qui figurent en partie droite des schémas de règle, le processus s'arrête en un nombre fini d'étapes.

Exemple : l'hypernotion declaration prelude sequence option entraine tout d'abord la création du schéma

declaration prelude sequence option :
    declaration prelude sequence ; .

ce qui amène à créer ensuite

declaration prelude sequence :
    chain of declaration preludes separated by sequence separators.

puis

chain of declaration preludes separated by sequence separators :
    declaration prelude ; declaration prelude, sequence separator,
    chain of declaration preludes separated by sequence separators.

Nous noterons $G'_p = (M', \sum, \Delta', \Gamma', S')$ la nouvelle pseudogrammaire.

## 2.2. Remplacement de certaines notions par leurs productions terminales

Lorsque dans un schéma de règle apparait à droite du deux-points une *notion* dont le nombre de productions terminales est borné, ce schéma peut être remplacé

par autant de nouveaux schémas que la notion a de productions terminales, chacun d'eux étant obtenu en remplaçant simplement dans le schéma d'origine la notion par chacune de ses productions terminales. Il est clair qu'en procédant ainsi, le langage engendré par la grammaire G n'est pas changé.

Parmi toutes les notions susceptibles de subir ce remplacement, seules celles qui sont citées ci-après seront prises en compte ; ainsi,

a) **letter e letter x letter i letter t** sera remplacé par **letter e symbol, letter x symbol, letter i symbol, letter t symbol** dans le schéma R.2.1.e,

b) **letter e** sera remplacé par **letter e symbol** dans le schéma R.5.1.2.1.h,

c) **letter** ξ, où ξ est mis pour **k, x, y, l, p, z, n, c, e, b, a, t,** ou **i,** sera de même remplacé par **letter** ξ **symbol** dans toute la syntaxe des notations de format (schémas R.5.5.1.a à R.5.5.7.c).

d) **digit X,** où X est mis pour **zero, one, two, four, six** ou **eight** sera remplacé par **digit X symbol,** aussi dans la syntaxe des notations de format.

Tous les remplacements cités permettront de regrouper toutes les règles de production engendrées par les schémas R.3.0.2.b, R.3.0.3.c,d et R.4.1.1.c,d, en quelques règles lors de la construction de la c-grammaire cherchée, et ceci sans perdre de précision pour les règles qui seront obtenues à partir des schémas concernés par ces remplacements. Nous noterons G' la nouvelle grammaire d'Algol 68 telle qu'elle est obtenue à l'issue des traitements précédents (paragraphe 2.1 et 2.2).

Remarque :

Chacune des transformations décrites aux paragraphes précédents peut être réalisée indépendamment des autres. Il est donc possible de les effectuer dans un ordre quelconque sans modifier le langage engendré par la grammaire G'. Cependant, pour conserver le maximum de précision dans la syntaxe "context-free" déduite de G', nous réaliserons ces transformations dans l'ordre où elles ont été décrites.

## 3 - OBTENTION DE LA C-GRAMMAIRE

Nous procéderons en deux étapes :

Dans une *première phase*, nous construirons l'ensemble des *non-terminaux* de

la c-grammaire cherchée en associant à chaque hypernotion constituant le membre gauche d'un schéma de règle de la nouvelle pseudogrammaire, un non-terminal de cet ensemble. Nous représenterons chacun de ces non-terminaux par une protonotion et nous utiliserons des formes françaises ; en effet nous pouvons profiter de cette application pour obtenir une c-grammaire, tout en utilisant la grammaire originale qui est à consonances Anglaises.

En pratique, lorsque l'hypernotion est une notion, le non-terminal associé est la traduction française [2] de cette notion ; par exemple, à standard prelude nous associons <prologue standard>. Dans d'autres cas nous supprimerons ce qui concerne les modes et modifications dans la traduction française de l'hypernotion ;

exemples : à MABEL identifier est associé <identificateur>
à SORTETY MOID unit est associé <unité>

Parfois, nous utiliserons les schémas astérisqués dont l'hypernotion droite engendre les mêmes notions que l'hypernotion donnée ;

exemples : à procedure with PARAMETERS MOID denotation est associé <notation de routine>
à row of character denotation est associé <notation de chaîne>

L'application ainsi définie n'est pas nécessairement injective et nous associerons à des hypernotions différentes le même non-terminal.

Ainsi, après les remplacements décrits aux paragraphes précédents, le schéma R.3.0.2.b (LETTER : LETTER symbol.) a été remplacé par les schémas suivants

letter a : letter a symbol.
letter b : letter b symbol.
...
letter z : letter z symbol.

Nous associerons aux hypernotions letter a, letter b, ..., letter z l'unique non-terminal <lettre> ; c'est pourquoi nous avons réalisé les remplacements du paragraphe 2.2. En multipliant ces regroupements, qui implique en définitive la représentation de plusieurs classes de notions par un seul non-terminal, la précision de la c-grammaire que nous obtiendrons diminuera ; (à la limite il est possible d'associer à toutes les hypernotions gauches de la pseudogrammaire un seul non-terminal). Ces regroupements permettent aussi de réduire en partie dès cette phase la c-grammaire cherchée ; en effet considérons le schéma suivant :

declaration prelude sequence option : declaration prelude sequence ; .

Nous avons vu qu'après traitement des constructions générales il donnait naissance aux schémas :

(1) declaration prelude sequence :
chain of declaration preludes separated by sequence separators.

(2) chain of declaration preludes separated by sequence separators :
declaration prelude ; declaration, sequence separator,
chain of declaration preludes separated by sequence separators.

Nous voyons dès maintenant que la règle (2) est inutile et que l'une des deux notions qui y figurent l'est aussi ; nous associerons donc à ces deux notions l'unique non-terminal <declaration en prologue en sequence> par exemple. Cette remarque est valable pour tous les schémas issus de R.3.0.1.d.

L'*axiome* de la c-grammaire cherchée sera le non-terminal associé à l'axiome de la grammaire G' ; ce dernier est **program**, auquel est par exemple associé <programme>.

Pour construire l'*ensemble des terminaux* de la c-grammaire nous associerons, de manière biunivoque cette fois, un terminal à chaque symbole au sens d'Algol 68, figurant dans la pseudogrammaire. Le nombre des symboles n'est pas borné, puisque toute protonotion se terminant par **symbol** est un symbole. Nous nous limiterons donc aux symboles utiles. Par ailleurs la grammaire G' n'est pas, en principe unique, car il est possible de lui ajouter des règles de production pour **indicant**, **dyadic indicant** et **monadic indicant** dont chaque production directe est un nouveau symbole. La c-grammaire obtenue sera, si l'on peut dire, aussi incomplète que G car elle ne comportera ni ces nouveaux symboles, ni les règles de production leur correspondant.

Il reste dans une *deuxième phase* à construire l'ensemble des *règles de production* de la c-grammaire cherchée.

Nous avons déjà vu au chapitre 1 que les schémas de règle étaient regroupés pour donner des schémas qui ont la forme suivante :

(3) $\gamma : \delta_1, \delta_2, \ldots, \delta_k ;$
$\delta_{k+1}, \delta_{k+2}, \ldots, \delta_{k+l}, \ldots ;$
$\delta_p, \delta_{p+1}, \ldots, \delta_{p+q}.$

avec $\gamma \in \Gamma'$ et $\delta_i \in \Delta'$ pour $1 \leq i \leq p+q$

Nous noterons h l'application qui a un élément de $\Gamma'$ associe un non-terminal de la c-grammaire cherchée et g celle qui a un élément de $\mathcal{C}$, ensemble des symboles, associe un terminal de cette même c-grammaire.

Pour tout $\delta_i \in \Delta' - \mathcal{C}$, nous considérerons les ensembles suivants :

- $\Gamma'_i = \{ \gamma \mid \gamma \in \Gamma', L(\gamma) \cap L(\delta_i) \neq \emptyset \}$

($L(\gamma)$ représentant par abus de notation l'ensemble des notions engendrées par $\gamma$ et d'une manière analogue $L(\delta_i)$ représentant l'ensemble des protonotions engendrées par $\delta_i$)

- $V_i = \{h(\gamma) \mid \gamma \in \Gamma'_i\} = \{v_{i1}, v_{i2}, \ldots, v_{ij}, \ldots, v_{ir}\}$

Pour tout $\delta_i \in \mathcal{C}$, nous poserons

- $V_i = \{g(\delta_i)\} = \{v_{i1}\}$

L'ensemble des règles de production de la c-grammaire est alors obtenu en créant pour tout schéma de la forme (3), les règles de production (avec mise en facteur de la variable)

$$h(\gamma) \rightarrow v_{1j_1}\ v_{2j_2} \ldots v_{kj_k} \mid v_{k+1\ j_{k+1}} \ldots v_{k+l\ j_{k+l}} \mid \ldots \mid v_{pj_p} \ldots v_{p+q\ j_{p+q}}$$

pour toutes les combinaisons possibles des éléments des ensembles $V_1, V_2, \ldots, V_{p+q}$.

Il est clair que l'ensemble des transformations ainsi décrites définissent un homomorphisme de c-grammaire entre la grammaire G' et la c-grammaire que nous avons construite.

Nous pouvons, pour rechercher les éléments des ensembles $\Gamma_i$, utiliser le système de renvois figurant dans la pseudogrammaire, à condition naturellement de modifier de façon convenable ces renvois au fur et à mesure que les transformations affectant la pseudogrammaire sont réalisées.

Nous avons cependant pensé qu'il était suffisant d'approcher les ensembles $\Gamma'_i$, c'est-à-dire de construire des ensembles $E_i$ tels que $\Gamma'_i \subseteq E_i$ et de leur associer des ensembles $U_i = \{h(\gamma) \mid \gamma \in E_i\}$. La grammaire "context-free" obtenue sera d'autant plus précise que les ensembles $E_i$ seront "plus proches" des ensembles $\Gamma'_i$ correspondants, c'est-à-dire qu'ils ne diffèreront que par un nombre

restreint d'éléments.

Pour approcher les ensembles $\Gamma'_i$ nous utiliserons les idées données au chapitre 1 pour améliorer la recherche des membres gauches ; nous supposerons de la même manière que les protonotions sont codées à l'aide de protonotions simples.

$\forall\ \omega \in \Gamma' \cup \Delta'$, nous noterons $\phi(\omega)$ l'élément de $\sum' \cup M$ qui termine $\omega$
et $\psi(\omega)$ l'élément de $\sum' \cup M$ qui commence $\omega$ ;

par extension nous noterons $\Phi(\omega) = \{\phi(w) \mid w \in L(\omega)\}$
et $\Psi(\omega) = \{\psi(w) \mid w \in L(\omega)\}$.

(a) Il est clair qu'une condition nécessaire pour que $L(\gamma) \cap L(\delta_i) \neq \emptyset$
est que $\Phi(\gamma) \cap \Phi(\delta_i) \neq \emptyset$.

Nous pouvons donc approcher l'ensemble $\Gamma'_i$ associé à l'hypernotion $\delta_i$ par l'ensemble $E_i^1$ défini comme suit :

$$E_i^1 = \{ \gamma \mid \gamma \in \Gamma',\ \Phi(\gamma) \cap \Phi(\delta_i) \neq \emptyset\}$$

Nous avons les inclusions suivantes $\Gamma'_i \subseteq E_i^1 \subseteq \Gamma'$, et comme nous l'avons vu au chapitre 1, la forme des notions anglaises implique que les ensembles $E_i^1$ ainsi obtenus constituent déjà une assez bonne approche des ensembles $\Gamma'_i$.

(b) De manière analogue, une condition nécessaire pour que $L(\gamma) \cap L(\delta_i) \neq \emptyset$
est que $\Psi(\gamma) \cap \Psi(\delta_i) \neq \emptyset$

Nous pouvons donc approcher l'ensemble $\Gamma'_i$ par l'ensemble $E_i^2$ :

$$E_i^2 = \{ \gamma \mid \gamma \in E_i^1,\ \Psi(\gamma) \cap \Psi(\delta_i) \neq \emptyset\} \text{ auquel est associé } U_i^2 = \{h(\gamma) \mid \gamma \in E_i^2\}$$

Les inclusions suivantes sont vérifiées $\Gamma'_i \subseteq E_i^2 \subseteq E_i^1 \subseteq \Gamma$.

Si card $U_i^2 = 1$, il est inutile de chercher à réduire le nombre d'éléments de $E_i^2$ et nous prendrons pour ensemble $E_i$ approchant $\Gamma_i$, l'ensemble $E_i^2$ ; dans le cas contraire, la réduction peut être poursuivie de façon assez asiée, si $\psi(\gamma)$ et $\psi(\delta_i)$ appartiennent à $\sum'$ ou à l'ensemble des métanotions dont les productions terminales sont de longueur 1 ; il suffit de priver $\gamma$, $\gamma \in E_i^2$, et $\delta_i$ de leur premier élément puis de réappliquer la réduction définie en (b).

La c-grammaire obtenue peut naturellement subir toutes les transformations qui conservent le c-langage qu'elle engendre : réduction, élimination des récursivités à gauche, mise sous diverses formes normales, suppression de certaines

ambiguités, etc. Il sera aussi utile de regrouper certaines règles, en particulie dans la syntaxe des déclareurs.

## 4 - DESCRIPTION DE LA SYNTAXE "CONTEXT-FREE" DU LANGAGE ETENDU PAR DES REGLES DE PRODUCTION

Les extensions sont décrites de manière assez informelle à l'aide de paranotions, de lettres représentant des paranotions ou des listes de paranotions et de phrases en langue anglaise. Ces dernières définissent des substitutions ou des omissions permises par rapport au langage strict. Les paranotions désignent de façon précise des ensembles de notions ; il est donc aisé d'associer aux paranotions utilisées les non-terminaux qui désignent les mêmes unités syntaxique Dans ce qui suit nous allons voir comment chaque extension peut être décrite par une ou plusieurs règles de production ; des non-terminaux supplémentaires sont introduits.

Certaines extensions relèvent d'un traitement à l'édition ; c'est le cas précisément de celle décrite au paragraphe R.9.1.a et qui traite des commentaires (il serait possible de la décrire à l'aide de règles de production, mais ce serait fastidieux et inutile).

L'extension du paragraphe R.9.2.a peut, par exemple, être décrite par les règles supplémentaires suivantes :

<déclaration d'identité> ::= <symbole tas en option> <déclareur effectif de mode>

<identificateur> <symbole devient source en option>

Les règles de production qui définissent <symbole devient source en option> sont les suivantes :

<symbole devient source en option> ::=
<symbole devient> <source> | vide

Elles se déduisent donc facilement du non-terminal donné ; dans la suite de ce chapitre nous ne ferons pas figurer de telles règles de production.

Celle du paragraphe R.9.2.b peut l'être par les règles :

<déclaration de mode> ::= <symbole structure> <indicateur de mode> <symbole égale>

<déclareur effectif de champs en paquet> |
<symbole union> <indicateur de mode> <symbole égale>
<piège tendu en paquet>

L'extension R.9.2.c est la plus complexe ; elle consiste à autoriser la mise en facteur de certains symboles ou de certains déclareurs dans des déclarations collatérales, des paramètres formels ou des déclarateurs de champs. Nous proposons d'ajouter, pour décrire l'ensemble de ces extensions, des règles de production dont nous ne donnerons que les suivantes :

<déclaration collatérale> ::=
<déclaration collatérale étendue ou déclaration unitaire en liste propre> |
<déclaration collatérale étendue>

<déclaration collatérale étendue> ::= <déclaration collatérale de mode> |
<déclaration collatérale de priorité> |
<déclaration collatérale d'identité> |
<déclaration collatérale d'opération>

<déclaration collatérale de mode> ::=
<symbole mode> <indicateur de mode symbole égale déclareur effectif de mode en liste propre> | <symbole structure> <indicateur de mode symbole égale déclareur effectif de champs en paquet en liste propre> | <symbole union> <indicateur de mode symbole égale piège tendu en paquet en liste propre>

<déclaration collatérale de priorité> ::=
<symbole priorité> <indicateur dyadique symbole égale marque de niveau en liste propre>

<déclaration collatérale d'identité> ::=
<déclareur formel de mode> <identificateur symbole égale paramètre effectif en liste propre> |
<symbole structure> <déclareur formel de champs en paquet> <identificateur> <symbole égale> <paramètre effectif> <symbole virgule>
<déclareur formel de champs en paquet identificateur symbole égale paramètre effectif en liste> <symbole virgule> <identificateur symbole égale paramètre effectif en liste> |

<symbole structure> <déclareur formel de champs en paquet> <identificateur> <symbole égale> <paramètre effectif> <symbole virgule> <déclareur formel de champs en paquet identificateur symbole égale paramètre effectif en liste> |
<symbole structure> <déclareur formel de champs en paquet> <identificateur> <symbole égale> <paramètre effectif> <symbole virgule> <identificateur symbole égale paramètre effectif en liste> |
<symbole structure> <déclareur effectif de champs en paquet> <identificateur> <symbole devient source en option> <symbole virgule> <déclareur effectif de champs en paquet identificateur symbole devient source en option en liste> <symbole virgule> <identificateur symbole devient source en option en liste> |
<symbole structure> <déclareur effectif de champs en paquet> <identificateur> <symbole devient source en option> <symbole virgule> <déclareur effectif de champs en paquet identificateur symbole devient source en option en liste> |
<symbole structure> <déclareur effectif de champs en paquet> <identificateur> <symbole devient source en option> <symbole virgule> <identificateur symbole devient source en option en liste> |

{des règles analogues aux précédentes sont aussi créées pour la mise en facteur du symbole *union* ; il suffit de remplacer structure par union et déclareur formel ou effectif de champs en paquet par piège tendu en paquet}

<symbole tas> <déclareur effectif de mode> <identificateur> <symbole devient source en option> <identificateur symbole devient source en option en liste>

<déclaration collatérale d'opération> ::=
<symbole opération> <affiche symbole égale paramètre effectif ou opérateur adique symbole égale notation de routine avec ou sans parenthèses en liste propre>

La mise en facteur dans les paramètres formels et les déclarateurs de champs ne pose pas de problèmes supplémentaires. Nous remarquerons que cette description est assez lourde.

Les extensions du paragraphe R.9.2.d sont décrites par

<paramètre effectif> ::= <paramètres formels en paquet> <forceur> |
<forceur neutre>

<source> ::= <paramètres formels en paquet> <forceur>

En ce qui concerne les extensions du paragraphe R.9.2.e, la première et la dernière sont déjà décrites par la grammaire "context-free" obtenue à l'issue du paragraphe 3 puisque cette dernière n'impose aucune contrainte sur la compatibilité des modes. Il reste donc la deuxième qui donne lieu à la création de la règle

<déclaration d'opération> ::=
<symbole opération> <opérateur adique> <symbole égale> <notation de routine avec ou sans parenthèses>

Enfin toutes les autres extensions se décrivent très facilement à l'aide de règles de production. Nous donnerons pour terminer ce chapitre les règles décrivant les instructions d'itération (paragraphes R.9.3. a, b, c) :

<proposition unitaire> ::=
<déclaration de variable controlée en option> <départ en option> <pas en option> <arrivée en option> <partie tant que en option> <symbole faire> <proposition unitaire>

<déclaration de variable controlée en option> ::=
<symbole pour> <identificateur> | vide

<départ en option> ::=
<symbole depuis> <proposition unitaire> | vide

<pas en option> ::=
<symbole pas> <proposition unitaire> | vide

<arrivée en option> ::=
<symbole jusqu'à> <proposition unitaire> | vide

<tant que en option> ::=
<symbole tant que> <proposition sérielle> | vide.

# CHAPITRE 3

## REPRESENTATION INTERNE DES DECLAREURS ET DES MODES.

Algol 68 permet de traiter des objets appelés valeurs. Les valeurs qui ont les mêmes propriétés sont regroupées en ensembles. Le langage distingue ainsi l'ensemble des nombres entiers, celui des nombres complexes, celui des valeurs logiques, celui des chaînes de caractères, etc... A chacun de ces ensembles est associé un *mode* qui rend compte des propriétés de cet ensemble. En Algol 68, le nombre de ces ensembles de valeurs n'est pas limité comme dans d'autres langages ; ceci implique l'utilisation d'un langage formel pour désigner ces ensembles. [1] définit un mode comme une production terminale de la métanotion MODE. Cette définition ne peut être que théorique car elle ne permet pas d'obtenir pratiquement les productions terminales de longueur infinie qui caractérise les *modes récursifs* (voir paragraphe 1.3).

En réalité les modes sont définis dans un programme par des déclareurs, c'est-à-dire par des suites finies de symboles obéissant à des règles d'écriture très précises. Cependant l'application qui à un déclareur associe le mode qu'il représente n'est pas injective. En effet l'introduction de *déclarations de mode*, grâce auxquelles les modes récursifs peuvent être définis par un nombre fini de symboles, implique la possibilité d'écrire un déclareur représentant un même mode de plusieurs façons différentes. Par exemple, dans le domaine de validité de la déclaration de mode

*mode compl = struct (réel rô, réel théta)*, les deux déclareurs suivants *rep [:] compl* et *rep [:] struct (réel rô, réel théta)* représentent le même mode. Par ailleurs la syntaxe du langage permet certaines libertés d'écriture pour les déclareurs spécifiant des *modes unis* (voir 1.2.5) ; ainsi les déclareurs

*union (réel, union (ent, bool))*, *union (bool, ent, réel)* et *union (union (réel, ent), union (ent, bool))* représentent tous les trois le même mode.

Il résulte de ces particularités que la comparaison symbole à symbole de deux déclareurs ne suffit pas pour conclure qu'ils représentent des modes

différents. Or il est nécessaire de pouvoir déterminer si deux déclareurs spécifient le même mode, notamment

- pour contrôler la validité des déclareurs de mode (apparentement des composantes d'union);
- pour identifier les opérateurs et traiter les modifications (voir chapitre 5) ;
- pour élaborer les relations de conformité.

Le but de ce chapitre est de définir une représentation interne des déclareurs, donc des modes, qui permette de comparer aisément les modes représentés par ces déclareurs. Dans une première partie nous préciserons la manière de construire les modes ; dans une deuxième partie nous décrirons les *systèmes d'équations* représentant les modes et la façon de les obtenir.

## 1 - LES MODES EN ALGOL 68

Certaines valeurs peuvent être traitées directement par calculateur automatique (c'est le cas le plus souvent des nombres entiers, des booléens, des caractères) ou bien sont d'un emploi très fréquent (les nombres réels). Ces valeurs sont considérées comme primitives ; il leur correspond des modes simples qui rendent compte de leurs propriétés définies dans le langage lui-même.

Algol 68 fournit en plus la possibilité de construire de nouveaux ensemble de valeurs à partir des valeurs primitives. Le mécanisme de construction obéit à cinq règles qui peuvent être appliquées de façon récursive. A chaque nouvelle classe de valeurs ainsi construites est associé un nouveau mode représenté par un ou plusieurs déclareurs. Dans tous les cas nous dirons qu'un déclareur spécifie un mode.

Par abus de notation nous désignerons souvent les modes par leurs déclareu

### 1.1. Modes des valeurs primitives

Ensemble de valeurs :

| | déclareur |
|---|---|
| {*vrai*, *faux*} (valeurs logiques) | *bool* |
| $\{n \mid n \in Z, \lvert n\rvert < m_1\}$ (entiers simple longueur) | *ent* |

$\{n \mid n \in Z, |n| < m_2$ avec $m_2 \geq m_1\}$ (double longueur) *long ent*

(suivant l'implémentation d'autres longueurs d'entiers peuvent être utilisables)

$\{x \mid x = m \times b^e, b \in \mathbb{N}, m \in Z, e \in Z, |m| < m_3, |e| < e_1\}$

(réels simple précision) *réel*

$\{x \mid x = m \times b^e, b \in \mathbb{N}, m \in Z, e \in Z, |m| < m_4,$

$m_4 \geq m_3, |e| < e_2, e_2 \geq e_1\}$ (réels double précision) *long réel*

(de même, suivant l'implémentation d'autres précisions de réels peuvent être utilisables)

$\{k \mid k \in K, |K| < p\}$ (caractères) *car*

{formats d'entrée-sortie} *format*

Remarque 1 :

Les formats sont des chaînes dont les propriétés sont limitées pour le programmeur ; ainsi il ne pourra ni les concaténer, ni en extraire ou modifier des éléments. Le fait de les considérer comme primitifs constitue un artifice de description.

Remarque 2 :

D'autres valeurs construites en principe à partir des valeurs primitives doivent être considérées en elles-mêmes comme primitives, car de même ces valeurs ne possédent que des propriétés limitées pour le programmeur. Les modes de ces valeurs sont ceux spécifiés par les déclareurs standards, c'est-à-dire définis par le langage, tels que *séma*, *fichier*, *bits*, *long bits*, *cat*, *long cat* etc...

## 1.2. Règles de construction de modes

Ces règles sont au nombre de cinq et elles peuvent s'appliquer récursivement ce qui entraîne que le nombre de mode n'est pas borné.

### 1.2.1. Première règle : repères

Pour tout mode μ, une variable à valeurs dans l'ensemble des valeurs de

mode μ est une nouvelle valeur de mode "repère de μ" ; ce mode est spécifié par le déclareur *rep m*, où *m* représente un déclareur spécifiant le mode μ.

Exemples : aux valeurs de modes *ent* et *rep réel*,
on associe respectivement les valeurs de mode *rep ent* et *rep rep réel*.

1.2.2. Deuxième règle : structures et valeurs structurées

Pour toute suite finie non vide de modes $\mu_1$, $\mu_2$, ..., $\mu_n$, un élément du produit cartésien des ensembles de valeurs de mode $\mu_i$ ($1 \leq i \leq n$), chacune de ces valeurs étant indexée par un identificateur $\sigma_i$, est une valeur appelée *valeur structurée*. Le mode d'une telle valeur est spécifié par le déclareur *struct (m1 σ1, m2 σ2, ..., mn σn)*, où *mi* représente un déclareur spécifiant le mode $\mu_i$. Les identificateurs qui sont appelés *sélecteurs de champs* font partie du mode d'une valeur structurée.

Exemples : - l'ensemble des valeurs complexes peut ainsi être caractérisé par le mode *struct (réel re, réel im)* et c'est un autre ensemble de valeurs que caractérise le déclareur *struct (réel rô, réel théta)*,

- en utilisant pour éléments, des valeurs non primitives les structures obtenues peuvent être plus riches :

*struct (rep bool, struct (rep ent rent, rep rep réel rrr))*

1.2.3. Troisième règle : tableaux ou valeurs multiples

Pour tout mode μ, une suite finie et ordonnée de valeurs de mode μ forme une valeur multiple. Le mode d'une telle valeur est "rang de μ" et est spécifié par [:] *m*, où *m* spécifie le mode μ. Si le mode μ est à son tour celui d'une valeur multiple, c'est-à-dire de la forme "rang de μ'", le mode "rang de rang μ'" sera spécifié par le déclareur [:,:] *m'*, où *m'* spécifie le mode μ'. Ceci revient à dire que le déclareur qui se trouve derrière un crochet fermé ne doit pas spécifier un mode commençant par "rang de".

Exemples de déclareurs de valeurs multiples : [:] *ent*;[:,:] *rep réel* ;
[:] *struct (réel re, réel im)*;[:] *rep* [:] *car*.

1.2.4. Quatrième règle : procédures

Pour toute suite finie et ordonnée de modes $\mu_1$, $\mu_2$, ..., $\mu_p$, μ, une procédure à p paramètres de modes $\mu_i$ ($1 \leq i \leq p$) et à résultat de mode μ est

une nouvelle valeur dont le mode est spécifié par *proc (m1, m2, ..., mp)m*, où *mi(m)* représente un déclareur spécifiant le mode $\mu_1(\mu)$. Si la procédure n'a pas de paramètres (p=0) le déclareur prend la forme *proc m* ; si la procédure ne fournit pas de valeur elle est "à résultat neutre" et son déclareur est alors de la forme *proc (m1, m2, ..., mp)* ; enfin en l'absence de résultats et de paramètres le déclareur se réduit à *proc*.

Exemples de déclareurs de procédures : *proc proc (réel) réel* ; *proc proc réel* ; *proc (rep ent, rep [:] struct (réel re, réel im)) bool* .

1.2.5. Cinquième règle : unions

Pour tous les modes obtenus en utilisant les quatre règles précédentes, un objet externe (c'est-à-dire une unité syntaxique comme un identificateur, une notation, une formule, etc...) de ce mode ne peut *posséder*, c'est-à-dire désigner, qu'une valeur de ce même mode. Les modes unis permettent de préciser qu'un objet externe peut posséder à un instant donné de l'élaboration une valeur dont le mode n'est pas complètement fixé par le texte du programme, mais pris parmi plusieurs modes possibles.

Etant donnés q modes $\mu_1$, $\mu_2$, ..., $\mu_q$ ($q \geq 2$) le mode "union de $\mu_1$ et $\mu_2$ ... et $\mu_q$" est spécifié par le déclareur *union (m1, m2, ..., mq)*. Par définition les modes $\mu_i$ ne doivent pas commencer par "union". Il n'existe aucune valeur d'un mode commençant par "union", mais à un instant donné la valeur possédée par un objet externe de mode union peut être de l'un des modes spécifiés par les déclareurs constituant le déclareur d'union.

Dans l'écriture des déclareurs d'union, l'ordre des déclareurs spécifiant les modes composants n'est pas significatif (*commutativité* de l'union) ; d'autre part si l'un des déclareurs *mi* spécifie à son tour un mode union le déclareur *union (m1, m2, ..., mq)* spécifie un mode où *mi* a été remplacé par la liste des déclareurs de mode le composant (*associativité* de l'union) ; enfin on peut écrire des déclareurs d'union avec des déclareurs spécifiant le même mode, ceci n'est pas significatif (*nilpotence* de l'union).

Exemples : *union (réel, ent)* ; *union (réel, ent, réel)* ; *union (réel, union (ent, réel))* ;
*union (rep bool, union ([:] car, struct (réel re, réel im)), proc)*.

## 1.3. Déclarations de modes

Dans un programme certains modes peuvent être utilisés très souvent et il serait fastidieux de réécrire plusieurs fois le même déclareur si celui-ci comporte beaucoup de symboles ; c'est pourquoi le langage permet de définir des abréviations à l'aide de *déclarations de mode*. Une telle déclaration a la forme : *mode* indicateur de mode = déclareur.

Exemple : *mode compl = struct (réel re, réel im)*

L'indicateur de mode est soit un nouveau symbole introduit par le programmeur soit un symbole déjà utilisé et qui est ainsi redéfini. En ce qui concerne les modes, cet indicateur est un déclareur équivalent au déclareur qui se trouve à droite du signe égal dans la déclaration, on peut donc utiliser un indicateur de mode dans un déclareur.

Exemple : l'indicateur *compl* défini ci-dessus peut être utilisé pour écrire le déclareur *union (bool, union (réel, compl))* qui spécifie le même mode que le déclareur *union (bool, union (réel, struct (réel re, réel im)))*.

Les déclarations de mode permettent donc d'alléger la tâche du programmeur. Un autre intérêt est la possibilité de définir des *modes récursifs*, c'est-à-dire des modes dont les déclareurs utilisent pour leur écriture un ou plusieurs déclareurs spécifiant les modes des premiers. De tels modes rendent compte des propriétés de valeurs qui sont des éléments de listes généralisées. La récursivité peut être directe, c'est-à-dire ne nécessiter qu'une déclaration de mode, ou indirecte, c'est-à-dire utiliser plusieurs déclarations de mode.

Exemples : - *mode livre = struct ([:] car auteur, rep livre suivant)*

- *mode h = struct (rep h père, rep f mère)*
  *mode f = struct (rep h père, rep f mère)*

La liberté laissée en ce qui concerne l'utilisation ou la non-utilisation des déclarations de mode entraîne la possibilité d'écrire des déclareurs spécifiant le même mode de plusieurs façon différentes. C'est le cas pour les déclareurs d'union ci-dessus. De façon analogue les deux déclarations précédentes peuvent être remplacées par :

*mode h = struct (rep h père, rep struct (rep h père, rep f mère) mère)*
*mode f = struct (rep struct (rep h père, rep f mère) père, rep f mère)*

mais aussi par la seule déclaration

*mode h = struct (rep h père, rep h mère)*

L'utilisation de déclareurs de mode différents pour spécifier un même mode peut se produire dans un même programme, le plus souvent dans le cas où les diverses parties d'un programme sont rédigées par des programmeurs différents. Il est alors nécessaire de pouvoir reconnaitre si les modes définis dans l'une des parties sont les mêmes que ceux utilisés dans une autre partie.

Pour résoudre le problème que nous venons d'énoncer, une condition préalable est de trouver une représentation interne des déclareurs qui permette la comparaison des modes qu'ils spécifient. C'est ce que nous allons examiner dans le paragraphe qui suit.

## 2 - REPRESENTATION INTERNE DES MODES

### 2.1. Principe

Nous avons vu que par définition même un mode non récursif est un mot appartenant à un certain c-langage. En effet un mode est une production terminale de la métanotion MODE figurant dans la métagrammaire $G_M$. La c-grammaire engendrant les modes non-récursifs a pour alphabet terminal l'ensemble $\sum$ des signes syntaxiques minuscules, pour alphabet non terminal l'ensemble des métanotions dont MODE dépend dans $G_M$, pour règles de production les règles de production pour ces métanotions et pour axiome MODE. De tels modes peuvent donc être considérés comme étant les solutions d'un système d'équations. Par extension les modes récursifs seront considérés eux aussi comme solutions d'un ensemble d'équations.

Chaque équation a la forme d'une déclaration de mode où une seule des cinq règles de construction de mode est utilisée dans le déclareur de droite. Les indicateurs de mode de ces déclarations jouent le rôle d'inconnues. Tous les modes utilisés explicitement ou non par le programmeur seront représentés par une inconnue. Ceci revient à définir tous les modes par des déclarations de mode. Ainsi les modes primitifs sont représentés par des inconnues définies par des déclarations du genre *mode xr = réel*. La forme de toutes ces équations

est à rapprocher d'une forme standard des c-grammaires où les règles de production sont de la forme : $\xi \rightarrow$ a u, où a est un terminal et u un mot éventuellement vide sur l'alphabet des non-terminaux.

De façon plus concise un système d'équations de mode S est un triplet S = (W, T, E) avec T ensemble fini de terminaux, $T = \bigcup_{i=1}^{5} T_i$

$T_1$ = {*bool*, *car*, *ent*, *réel*, *format*, *long cat*, *bits*, ..., *neutre*}
($T_1$ représente l'ensemble des modes primitifs et de certains modes standards).

$T_2$ = {*proc*, *rep*, *rang*}

$T_3$ = {*procavec*, *union*}

$T_4$ = {*struct*}

$T_5$ = {*a*, *b*, *c*, ..., *y*, *z*, *0*, *1*, ..., *g*}

W ensemble des inconnues, $W = W_m \cup W_s$, $W_m$ étant l'ensemble des inconnues représentant des modes et $W_s$ celui des inconnues représentant des sélecteurs de champs.

E ensemble des équations, $E = \bigcup_{j=1}^{5} E_j$

$E_1 \subseteq W_m T_1$ (définition des modes primitifs)

$E_2 \subseteq W_m T_2 W_m$ (procédures sans paramètres, variable et tableaux)

$E_3 \subseteq W_m T_3 W_m^2 W_m^*$ (procédures avec paramètres et unions)

$E_4 \subseteq W_m T_4 (W_m W_s)^+$ (structures)

$E_5 \subseteq W_s (T_5 W_s \cup T_5)$ (définitions des sélecteurs)

Dans un programme on ne peut écrire qu'un nombre fini de déclareurs, le nombre des modes intervenant dans un même programme est donc fini ; il en résulte que le système d'équations représentant les modes d'un programme a un nombre fini d'équations.

Remarquons que nous avons introduit le terminal *neutre* qui sert à préciser l'absence de résultat pour une valeur de mode procédure. Le genre neutre est donc considéré comme un mode primitif. (Ce n'est d'ailleurs qu'au niveau des déclareurs que le "neutre" est représenté par le vide). D'autre part nous avons introduit le terminal *rang* pour représenter les modes des valeurs multiples (en effet l'utilisation des crochets, virgules et deux points n'est utile que pour préciser les bornes). Nous avons pensé à introduire un terminal différent pour chaque nombre de dimension de valeurs multiples ; cette méthode n'a pas été

adoptée car elle ne permettait pas de représenter aisément tous les modes d'un programme, en particulier, dans ce cas, ceux de toutes les tranches déduites des valeurs multiples. Nous avons encore distingué le cas des procédures avec paramètres du cas des procédures sans paramètres en utilisant deux terminaux différents : *proc* et *procavec* (cette distinction rendra plus aisée la vérification de la condition sur les modes). Enfin nous remarquerons que l'omission des parenthèses n'introduit aucune ambiguité dans la signification des équations.

Exemple de système d'équation (voir représentation pratique au paragraphe 2.2)

$x_b$ *bool*
$x_e$ *ent*
$x_r$ *réel*
$x_c$ *struct* $x_r$ $x_1$ $x_r$ $x_2$
$x_1$ *r* $x_3$
$x_3$ *e*
$x_2$ *i* $x_4$
$x_4$ *m*
$x_5$ *rep* $x_r$
$x_6$ *union* $x_r$ $x_c$

## 2.2. Représentation pratique

Les terminaux de l'ensemble $T_1$ et ceux des ensembles $T_2$, $T_3$ et $T_4$ sont codés par des entiers négatifs. Les sélecteurs de champs ne sont pas représentés par des inconnues et les équations associées mais directement par l'entier les codant à l'issue d'une phase d'édition.

Les membres droits des équations sont placés dans un tableau unidimensionnel à la suite les uns des autres (tableau appelé "système"). Un vecteur d'accès contient les adresses des seconds membres des équations dans le tableau système et les inconnues sont représentées par les adresses des éléments de ce vecteur. Les indicateurs de modes définis dans le programme sont placés dans une table qui fournit pour chacun d'eux le mode associé. (Nous supposons que l'identification des indicateurs de mode a été réalisée). Nous assimilerons à des indicateurs de mode les symboles constituant les déclareurs des modes primitifs.

Par ailleurs, afin de faciliter le test d'égalité (voir chapitre 4, paragraphe 2), nous placerons le nombre d'inconnues des seconds membres commençant

par les symboles *procavec*, *union* et *struct* entre ces symboles et la première inconnue. Pour les seconds membres commençant par *struct* les sélecteurs de champs seront tous placés en tête et non devant chaque inconnue représentant le mode des champs. Enfin, pour les modes des valeurs multiples et structurées, nous réserverons un certain nombre de places à la suite du second membre correspondant, de manière à pouvoir y placer les inconnues représentant les modes des repères aux composants de ces valeurs le cas échéant. Ceci permettra de trouver aisément, lors de l'analyse du mode d'une sélection sur repère de valeur structurée ou d'une tranche sur un repère de valeur multiple, le mode du sous-nom correspondant (voir chapitre 5, traitement de toutes les modifications).

Le schéma 1 (voir ci-contre) représente un système d'équations. Les modes primitifs spécifiés par *bool*, *ent* et *réel* y sont représentés ainsi que le mode standard *compl*. Pour ce dernier la table d'indicateurs fournit l'adresse, dans le vecteur d'accès du mot où se trouve l'adresse de la représentation du second membre définissant le mode *compl*. (Les relations adresse-contenu sont représentées par des flèches). Ce second membre est constitué dans l'ordre de la représentation de *struct*, de l'entier 2 car ce mode comprend deux champs, des codes internes des sélecteurs *re* et *im*, des adresses de l'élément du vecteur d'accès pointant sur la représentation du second membre de l'équation définissant le mode *réel*, enfin des adresses de l'élément du vecteur d'accès pointant sur la représentation du second membre associé à *rep réel*. Ces deux dernières adresses sont significatives car, dans le schéma, le mode des variables complexes est représenté (*rep compl*). Les modes *f* et *h* du paragraphe 1.3, ainsi que les modes implicitement définis par eux, sont aussi représentés.

## 2.3. Construction du système à partir des déclareurs

Pour simplifier la description de l'algorithme de construction de la représentation interne du système d'équations nous supposerons qu'il ne permet de traiter que des déclareurs virtuels et des déclarations de modes sans extensions ; (nous supposerons que les déclareurs des déclarations de mode sont aussi virtuels). Ceci permet de ne pas prévoir l'ouverture de régions ou l'utilisation de déclareurs (pour les générateurs globaux) dans les paires de bornes de déclareurs de valeurs multiples. D'autre part seuls sont utilisés pour ces déclareurs des crochets et non les parenthèses.

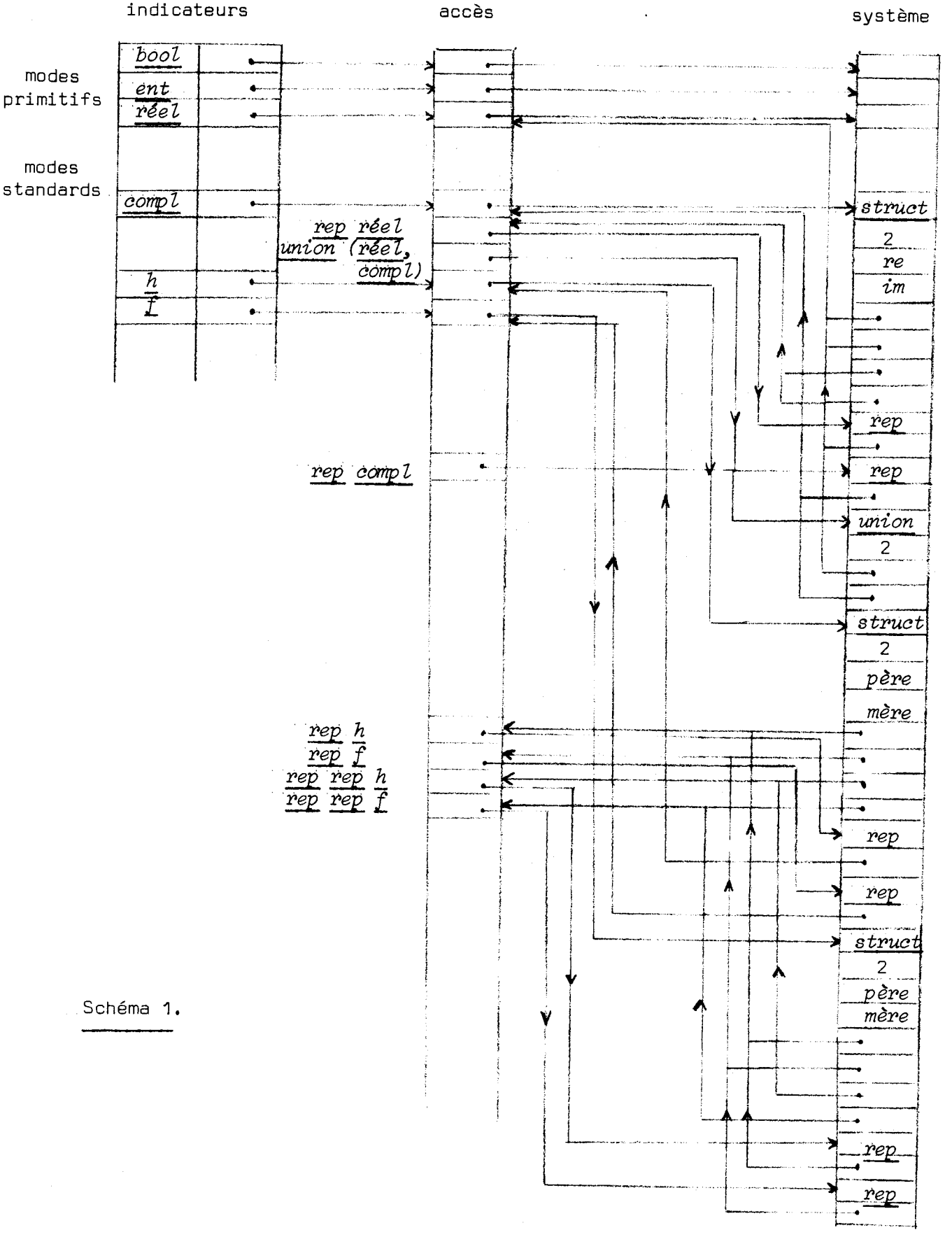

indicateurs
accès
système
modes primitifs
modes standards
bool
ent
réel
compl
rep réel union (réel, compl)
h
f
rep compl
rep h
rep f
rep rep h
rep rep f
struct
2
re
im
rep
rep
union
2
struct
2
père
mère
rep
rep
struct
2
père
mère
rep
rep

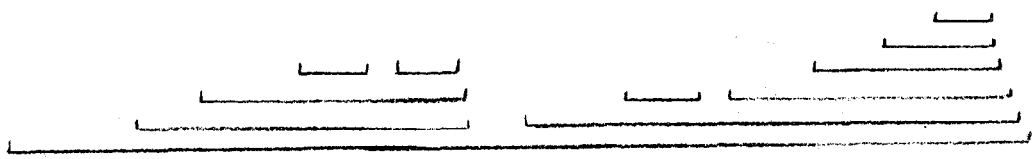
Schéma 1.

Certaines extensions pourraient aisément être traitées par le même algorithme, en particulier les mises en facteurs de déclareurs ou de symboles tels que *struct*, *mode*, etc. Le traitement des déclarations d'identité contractées et des paramètres formels des notations de routine entrainerait de légères modifications. Tout ceci dépend aussi un peu de l'existence et de la nature de l'analyse syntaxique réalisée avant la construction du système.

L'algorithme utilise un automate à pile qui analyse les déclareurs et les déclarations de mode. Les déclareurs appartiennent à un langage parenthésé ce qui permet une réduction de proche en proche, et leur remplacement progressif par des inconnues du système d'équations. Cette réduction est analogue à celle des expressions arithmétiques complètement parenthésées. Elle consiste à rechercher la paire de parenthèses la plus interne, la notion de parenthèse doit être prise au sens large, en déduire le second membre de l'équation qui définira le même mode que celui spécifié par le déclareur correspondant à cette paire de parenthèses, à remplacer ce dernier par l'inconnue définie par cette équation et à rechercher de nouveau la paire de parenthèses la plus interne.

Exemple montrant la structure parenthésée d'un déclareur

*struct (rep proc (réel) réel x, union (bool, proc rep [:] ent)y)*



Rappelons que les déclareurs (virtuels) sont de la forme

*proc*
*proc D*
*rep D*
*proc (D, D, ..., D)*
*proc (D, D, ..., D) D*
*[:, :, ..., :] D*
*struct (D* $\sigma_1$*, D* $\sigma_2$*, ..., D*$\sigma_p$*)*
*union (D, D, ..., D)*

'indicateur' (y compris les déclareurs des modes primitifs), où *D* représente un déclareur quelconque et les $\sigma_i$ des sélecteurs de champs ; les déclarations

de mode sont elles de la forme '*mode* indicateur = *D*'. Remarquons que les déclareurs commençant par le symbole *long* sont considérés comme des indicateurs de mode et comme tels ont du être traités lors d'une phase d'édition.

La construction du système d'équations se fait à la compilation et nécessite un balayage du texte source. La partie du système correspondant aux modes primitifs et standards est supposée préexister. Au cours du balayage chaque symbole permettant d'écrire un déclareur donne lieu aux traitements suivants :

Parenthèses ouvrantes :

i) lorsque le symbole lu joue le rôle d'une parenthèse ouvrante (*rep*, *struct*, *union* et () le code correspondant est placé au sommet de la pile. De même les codes des sélecteurs et celui du symbole égal des déclarations de mode est mis en pile.

ii) lorsque le symbole lu est *proc*, le symbole suivant permet de distinguer le cas où *procavec* est empilé de celui où *proc* le sera. Dans ce dernier cas le symbole suivant *proc* permet de déterminer si le déclareur en cours d'analyse est celui d'une procédure sans résultat ; si telle est la situation l'inconnue correspondante (qui définit le "mode neutre") est empilée. A chaque fois que le sommet de pile est une inconnue, une séquence de réduction est exécutée en vue de tenter l'écriture d'une nouvelle équation.

le traitement des crochets et des virgules placées entre crochets provoque la mise en pile d'un nombre convenable de *rang*.

Indicateurs :

iii) lorsque le symbole lu est un indicateur de mode, deux cas peuvent se présenter :

- cet indicateur a déjà été rencontré, il a donc déjà été associé à une inconnue définissant le mode correspondant et par conséquent cette inconnue est empilée ce qui entraîne l'exécution de la séquence de réduction ;
- cet indicateur n'a pas été encore rencontré ; dans ce cas une nouvelle inconnue est créée sans lui associer le membre droit d'une équation mais en marquant dans le vecteur d'accès le fait que cette inconnue n'est pas encore définie (en y plaçant par exemple la valeur 0) ; la table des indicateurs est complétée avec cette nouvelle inconnue et celle-ci est placée dans la pile ce qui entraîne aussi la séquence de réduction.

Parenthèses fermantes :

iv) lorsque le symbole lu est une parenthèse fermante ()), le symbole placé en pile

"avant" la parenthèse ouvrante associée permet de préciser si le déclareur en cours d'analyse est celui d'une procédure ; dans ce cas, comme en ii), le symbole suivant permet de déterminer la présence ou l'absence de résultat et les conclusions sont les mêmes ; dans les autres cas il s'agit du déclareur d'une structure ou d'un mode uni, une nouvelle équation est donc écrite dans le système en utilisant les éléments du sommet de la pile (voir d)). Après chaque écriture d'une équation l'inconnue correspondante est empilée ce qui entraîne l'exécution de la séquence de réduction.

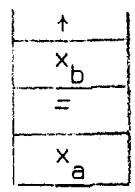
v) les autres symboles lus ne donnent lieu à aucun traitement en ce qui concerne la construction du système d'équations représentant les modes.

Séquence de réduction :

Après mise en pile d'une inconnue les cas suivants peuvent se présenter :

a) la pile ne comporte plus qu'un élément ; cet élément est alors nécessairement une inconnue ; le déclareur traité ne fait partie ni d'un autre déclareur ni d'une déclaration de mode et il peut donc être, du moins en ce qui concerne les modes, remplacé par l'inconnue qui se trouve au fond de la pile. Ce cas se présentera en particulier dans les déclareurs des déclarations d'identité et des générateurs ce qui permettra de leur associer une inconnue (voir identification des opérateurs). Pour les générateurs il faudra tenir compte du fait que leurs modes commencent toujours par "repère".

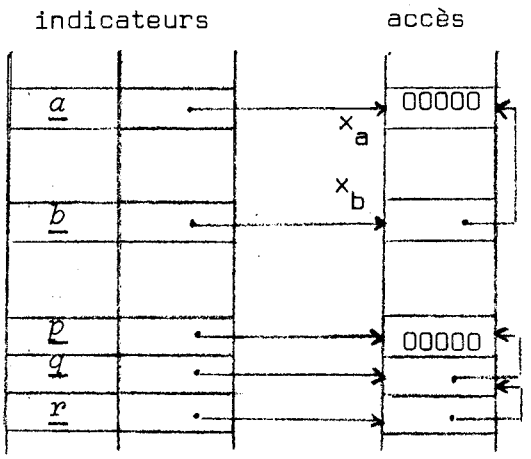
b) le sommet de pile est précédé d'un signe égal (cas d'une déclaration de mode) ; la pile a alors la configuration suivante où $x_a$ et $x_b$ représente des inconnues.




i) si $x_a$ est définie, c'est-à-dire qu'il lui correspond un second membre dans le système, il y a erreur car l'indicateur a déjà été défini par une déclaration de mode.

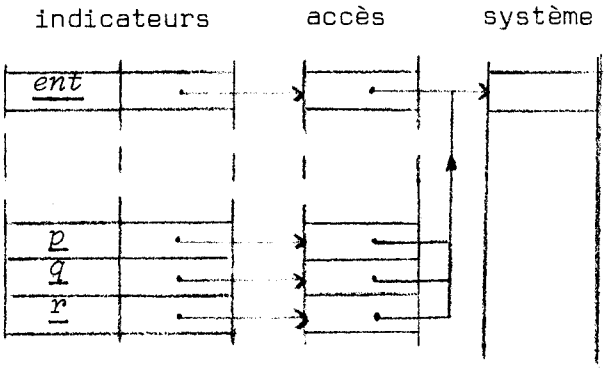
ii) si $x_a$ est non-définie et si $x_b$ est aussi non-définie les éléments correspondants du vecteur d'accès sont chainés ; cette situation ne peut se produire que dans le cas d'une déclaration de la forme *mode* *a* = *b* où *b* n'est pas encore défini.

Exemple : une déclaration collatérale de la forme *mode* *p* = *q*, *mode* *q* = *r* donnera après analyse de *r* la configuration ci-contre.

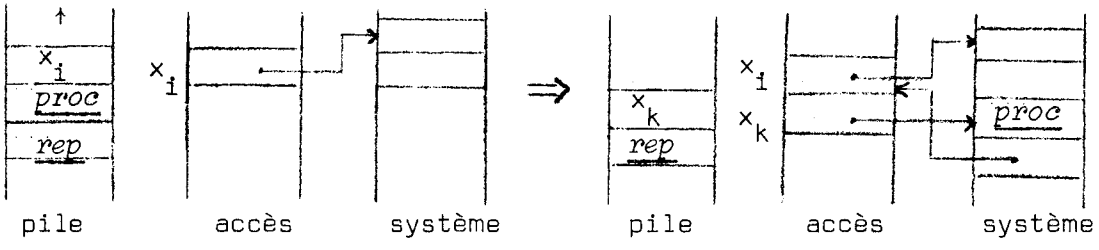



iii) si $x_a$ est non-définie et si $x_b$ est définie l'accès non défini est remplacé par celui correspondant à $x_b$ en tenant compte du chaînage.

Exemple : après la dernière déclaration de ii) et après le traitement de la déclaration suivante *mode* *r* = *ent*, la configuration ci-contre est obtenue.

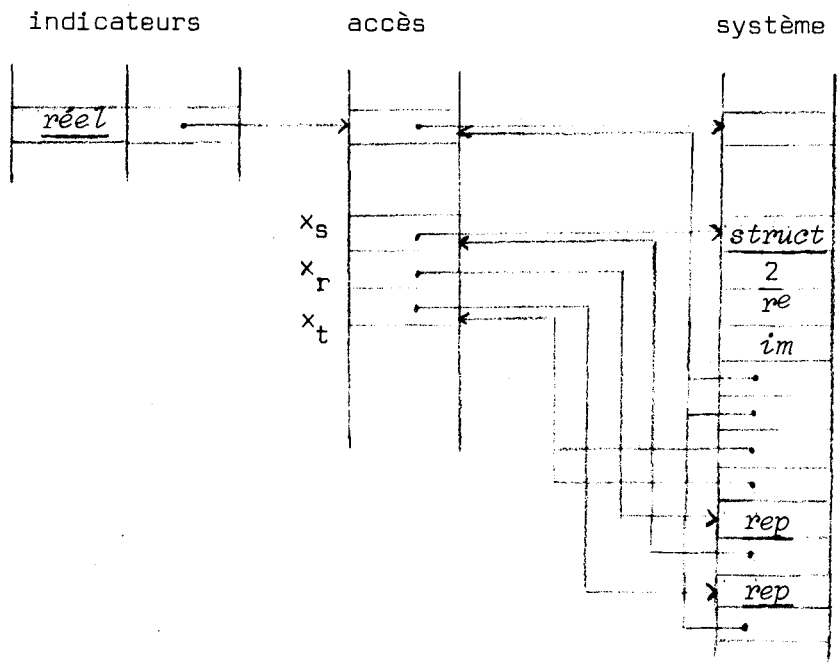



c) le sommet de pile est précédé de *rep*, *proc* ou *rang* ; dans ce cas le membre droit de l'équation correspondante est écrit dans le système, l'inconnue associée à ce membre droit est créée et remplace les deux éléments du sommet de pile. On réitère alors la séquence de réduction. On trouvera ci-dessous l'aspect de la pile lors de l'analyse de *rep* *proc* *m*.




En outre si le sommet de pile définit le mode d'une valeur structurée ou multiple et si l'élément précédant ce sommet est *rep*, le membre droit qui définit le mode structure ou rang est complété par les inconnues représentant les modes des sous-noms associés au repère de structure ou de tableau.

Dans le cas de l'analyse de *rep compl* on obtiendrait :




d) le sommet de pile est précédé d'une parenthèse fermante ; c'est le cas d'une procédure avec paramètres. Le membre droit de l'équation associée est écrit dans le système en utilisant les éléments qui se trouvent au sommet de la pile, l'inconnue correspondante est créée et les éléments de la pile sont dépilés jusqu'au symbole *proc* qui est remplacé par cette inconnue ; la séquence de réduction est donc de nouveau exécutée.

On trouvera en annexe une description plus précise de l'algorithme sous forme d'un programme Algol 68. La construction ainsi décrite pourrait être réalisée en même temps qu'une analyse ascendante en exploitant les résultats de cette dernière.

Le système obtenu est une représentation des déclareurs d'un programme. Il définit donc tous les modes utilisés dans un programme. L'objet du chapitre suivant est de s'assurer que certaines conditions sur l'écriture des déclareurs et des déclarations de mode sont vérifiées. Nous verrons aussi comment en déduire une représentation canonique des modes.

# CHAPITRE 4

## CONTROLE STATIQUE DES MODES ET DES DECLAREURS.

## REPRESENTATIONS CANONIQUES DES MODES.

Le langage impose trois conditions sur les modes et les déclareurs. Ces conditions, qui ne peuvent être décrites à l'aide d'une c-grammaire, s'énoncent brièvement comme suit :

(C1) *les sélecteurs de champs* d'un déclareur de structure doivent être *tous différents* (R.4.4.3.e),

(C2) les modes des composantes d'une union ne doivent pas être *apparentés deux à deux* (R.4.4.3.d), et

(C3) *l'indicateur de mode d'une déclaration de mode* ne doit pas être *montré* par le déclareur effectif de cette déclaration (R.4.4.4.c).

Ces conditions sont destinées, soit à éviter des ambiguités, soit à éliminer certains modes récursifs qui caractérisent des valeurs un peu trop encombrantes en pratique.

Le contrôle du non apparentement des composantes d'union nécessite la mise en oeuvre d'un algorithme permettant de déterminer si deux déclareurs spécifient le même mode. En effet, par définition, deux modes sont apparentés s'ils peuvent être obtenus par certaines *modifications* à partir d'un *même mode* (voir paragraphe 1.5).

Nous avons vu que certaines libertés d'écriture étaient autorisées par la syntaxe du langage pour les déclareurs d'unions. Pour simplifier le test d'égalité de modes, les unions sont *développées* ce qui élimine le problème posé par la multiplicité d'écriture des déclareurs utilisant l'associativité. Ce développement consiste à remplacer, dans le deuxième membre d'une équation commençant par le symbole *union*, les inconnues dont les seconds membres associés commencent aussi par *union*, par les inconnues apparaissant dans ces seconds membres ; le processus est itéré jusqu'à ce que les inconnues constituant le second membre d'une équation commençant par *union* représentent des modes non-

unis.

Exemple : le déclareur *union (réel, union (ent, bool))* est représenté par l'inconnue $x_u$ dans le système d'équations suivant :

$x_b$ *bool*
$x_e$ *ent*
$x_r$ *réel*

$x_u$ *union* $x_r$ $x_v$
$x_v$ *union* $x_e$ $x_b$

Après développement il est représenté par la même inconnue $x_u$ dans le nouveau système :

$x_b$ *bool*
$x_e$ *ent*
$x_r$ *réel*

$x_u$ *union* $x_r$ $x_e$ $x_b$

Le développement des unions nécessite à son tour le contrôle de la condition (C3) sur les déclarations de mode. En effet, si cette condition est vérifiée le traitement décrit au paragraphe précédent s'arrête en un nombre fini d'itérations. Ce ne serait pas le cas pour les équations associées aux déclareurs *x* et *a* définis par les déclarations de modes suivantes :

*mode x = union (a, bool, car) ;*
*mode a = union (x, ent) ;*

Nous verrons plus loin que l'algorithme qui permet de vérifier la condition (C2) nécessite aussi que (C3) soit vérifiée pour s'arrêter en un nombre fini d'étapes.

Le test d'égalité de modes se trouve simplifié si le système d'équations est réduit ; c'est-à-dire que, dans le cas où, pour chaque mode représenté par le système, il n'existe qu'une seule équation définissant ce mode, une simple comparaison des inconnues représentant deux modes permet de déterminer si ces derniers sont égaux. La réduction du système sera aussi utile lors de l'identification des opérateurs et du traitement des modifications (voir chapitre 5).

Enfin tous ces contrôles ne présentent un intérêt que dans la mesure où le système d'équations n'est pas entaché d'erreurs à la suite du non-respect de la condition (C1) sur les sélecteurs des déclareurs de structures.

Il résulte de tout ce qui précède que nous faisons, à partir de la représentation du système d'équations obtenue au chapitre 3, les vérifications et

traitements suivants dans l'ordre où ils sont énoncés :

- contrôle de (C1) (voir paragraphe 1.1),
- contrôle de (C3) (voir paragraphe 1.2),
- développement des unions (voir paragraphe 1.3),
- réduction du système d'équations (voir paragraphe 1.4),
- contrôle de (C2) (voir paragraphe 1.5).

Nous décrirons ensuite l'algorithme qui permet de déterminer si deux déclareurs spécifient des modes égaux. Enfin nous proposerons deux représentations canoniques des modes obtenues à partir de la représentation du système d'équations telle qu'elle résulte des traitements précédents.

## 1 - CONTROLE STATIQUE DES MODES ET DES DECLAREURS

### 1.1. Vérification de la condition sur les sélecteurs

Cette condition est très facile à vérifier puisqu'elle consiste à comparer deux à deux les codes des sélecteurs qui se trouvent dans les seconds membres des équations définissant des modes structurés. Etant donnée la représentation adoptée, cette vérification ne nécessite qu'un double balayage classique d'une partie de ces seconds membres. Cette condition peut d'ailleurs être vérifiée à la construction même du système d'équations à chaque fois qu'une équation dont le second membre commence par *struct* est écrite dans le système, ceci afin de ne pas poursuivre la compilation d'un programme erroné.

### 1.2. Véfification de la condition sur les déclarations de mode

Une partie de la structure du système d'équations peut être représentée par un graphe en associant à chaque équation un sommet du graphe, et en reliant par des arcs chaque sommet s aux sommets associés aux équations définissant les inconnues qui figurent dans le second membre de l'équation associée à s.

Exemple : les déclarations de mode

*mode a = struct (rep a u, rep b v)*

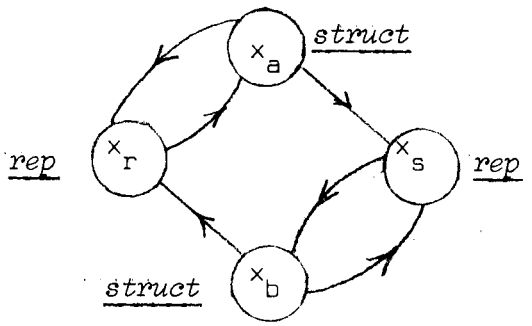
et *mode b = struct (rep a u, rep b v)*

définissent des modes dont les équations sont :

$x_a$ *struct* $\times_r u \times_s v$

$x_r$ *rep* $x_a$
$x_b$ *struct* $x_r$ $u$ $x_s$ $v$
$x_s$ *rep* $x_b$

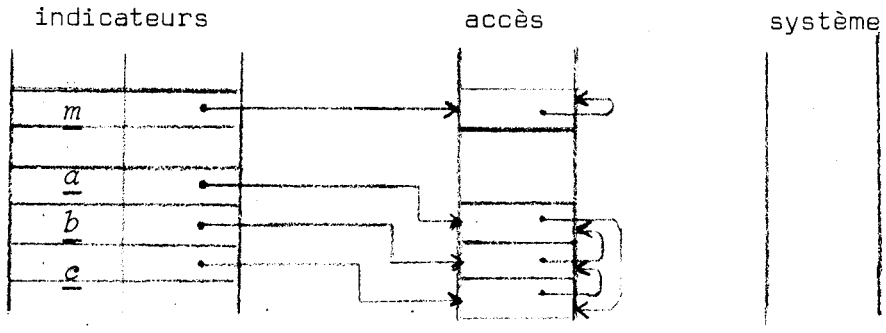
le graphe associé à ces équations peut être représenté par :




La condition sur les déclarations de mode peut alors s'exprimer de la façon suivante :
pour tout sommet associé à l'équation définissant le mode d'un indicateur de mode, tous les circuits passant par ce sommet doivent être tels qu'en les parcourant on rencontre

- soit un sommet dont le second membre de l'équation associée commence par *procavec*,
- soit un sommet dont le second membre de l'équation associée commence par *rep* (resp. *struct*) puis un sommet dont le second membre de l'équation associée commence par *struct* (resp. *rep*).

Cet énoncé diffère légèrement de celui qui figure en R.4.4.3.e car il ne tient pas compte des déclarations de mode "cycliques" de la forme *mode* $m$ = $m$ ou encore *mode* $a$ = $b$, *mode* $b$ = $c$, *mode* $c$ = $a$. En effet, à la construction de la représentation du système d'équations nous avons éliminé les déclarations de mode et de telles déclarations cycliques produisent des inconnues qui ne sont définies par aucune équation. Dans le cas des déclarations qui se trouvent ci-dessus, le résultat est le suivant :



La vérification consiste donc dans un premier temps à s'assurer que tout indicateur représente un mode défini par le second membre d'une équation. Ceci est aisément réalisable car dans le cas contraire, à l'adresse représentant l'inconnue associée à un indicateur de mode se trouve une adresse qui ne pointe pas dans le tableau des seconds membres mais dans celui des accès.

Dans un deuxième temps, un algorithme d'exploration du graphe associé à l'équation définissant le mode d'un indicateur permet de vérifier si ce dernier satisfait à la condition (C3). Cet algorithme utilise une pile pour ranger des doublets formés d'une inconnue et d'un entier indiquant si, sur le chemin qui relie le sommet associé à cette inconnue à celui associé à l'inconnue de départ, a été rencontré un sommet associé à une équation dont le second membre commence par *rep* ou par *struct* ou bien si un tel sommet n'a pas été rencontré. Cette pile contient des doublets qui doivent être examinés. D'autre part les inconnues rencontrées au cours du cheminement dans le graphe peuvent être marquées (par exemple au moyen d'un bit non utilisé dans les éléments constituant le tableau d'accès).

Phase d'initialisation :

Les inconnues éventuellement marquées, par exemple à la suite de la vérification de la condition sur un autre indicateur, sont d'abord "démarquées". Ensuite un doublet constitué de l'inconnue $x_0$ associée à l'indicateur faisant l'objet de la vérification et de l'entier indiquant que ni *rep* ni *struct* n'ont été rencontrés, est placé dans la pile.

Le processus suivant est alors itéré jusqu'à ce que la pile soit vide:

Le doublet qui se trouve au sommet de la pile est dépilé. L'inconnue correspondante, l'équation qui la définit et l'entier de ce doublet sont considérés ; si l'inconnue considérée est marquée le processus est immédiatement itéré ; dans le cas contraire, l'inconnue considérée est marquée,

puis a) si elle représente un mode primitif ou standard ou le mode d'une procédure avec paramètres, le processus est itéré (en effet dans le premier cas aucun circuit ne peut passer par le sommet associé à l'inconnue considérée, et dans le second cas, le second membre de l'équation associée à l'inconnue considérée commence par *procavec* et par conséquent en parcourant tous les circuits qui passent par le sommet correspondant et par le sommet associé à l'indicateur on rencontre nécessairement un sommet associé à un second membre commençant par *procavec*) ;

b) si elle représente le mode d'une valeur structurée (resp. d'un repère), si l'entier considéré indique qu'aucun *rep* (resp. *struct*) n'a été rencontré et si une des inconnues du second membre de l'équation considérée est identique à l'inconnue considérée alors *la condition sur les déclarations de mode n'est pas vérifiée* ; si la dernière condition n'est pas vérifiée, alors chacun des doublets formés d'une inconnue du second membre de l'équation considérée et de l'entier indiquant qu'un *struct* (resp. *rep*) a été rencontré sont empilés et le processus est itéré ;

c) si elle représente le mode d'une valeur structurée (resp. d'un repère), et si l'entier considéré indique qu'un *rep* (resp. *struct*) a déjà été rencontré, alors le processus est itéré (en effet en parcourant tous les circuits qui passent par le sommet associé à l'inconnue considérée et par le sommet associé à l'indicateur, on rencontre nécessairement deux sommets dont l'un est associé à un second membre commençant par *rep* et l'autre à un second membre commençant par *struct*) ; l'algorithme doit être poursuivi car il peut exister d'autres circuits passant par le sommet associé à l'indicateur faisant l'objet de la vérification.

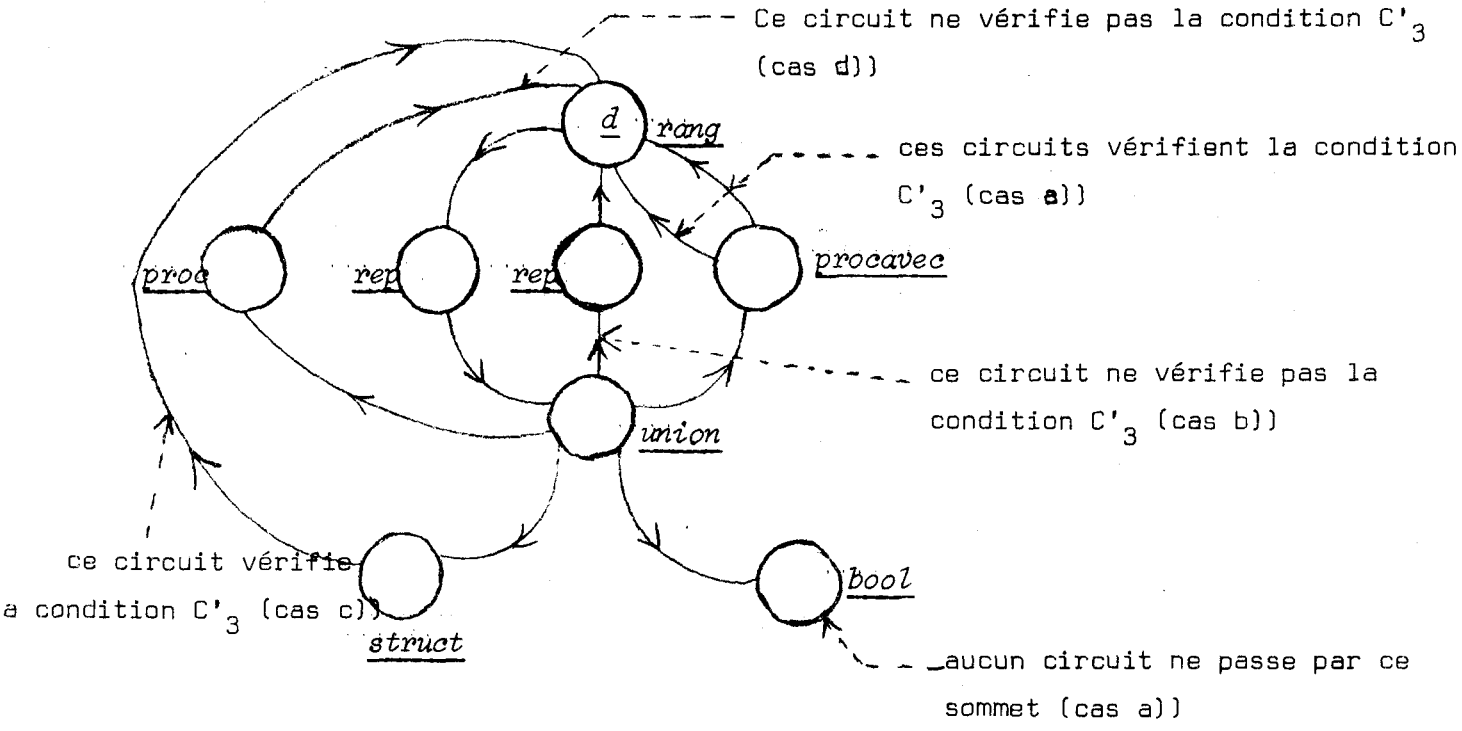
d) Dans tous les autres cas,
si une des inconnues du second membre de l'équation considérée est identique à l'inconnue considérée alors *la condition sur les déclarations de mode n'est pas vérifiée* ; dans le cas contraire,
chacun des doublets formés d'une inconnue du second membre de l'équation considérée et de l'entier considéré est placé dans la pile et le processus est itéré.

La condition sur les déclarations de mode est *vérifiée* si la *pile* est *vide* lors d'une nouvelle itération du processus.

Exemple : au mode de l'indicateur *d* dans la déclaration de mode

*mode d = [1:2] rep union (rep d, proc (d) d, bool, struct (d id), proc d)*

peut être associé le graphe qui figure ci-contre ; certains des circuits vérifient la condition $C'_3$ énoncée au début du paragraphe 1.2, d'autres ne la vérifie pas.




On trouvera en annexe une description plus précise de cet algorithme sous forme d'une partie de programme écrit en langage Algol 68.

1.3. Développement des unions

Ce traitement nécessite, pour chaque inconnue définissant un mode union, la mise en oeuvre de l'algorithme décrit ci-après. Cet algorithme utilise une pile en raison de la récursivité du processus ;

Phase d'initialisation :

Lors de cette phase, les inconnues constituantes du second membre commençant par le symbole *union* sont placées dans la pile et une variable booléenne est initialisée à faux. Cette variable permettra de ne pas créer de nouvelle équation, plus précisément de nouveau second membre, dans le cas où celà n'est pas nécessaire, c'est-à-dire lorsqu'aucune de ces inconnues ne représente à son tour un mode union.

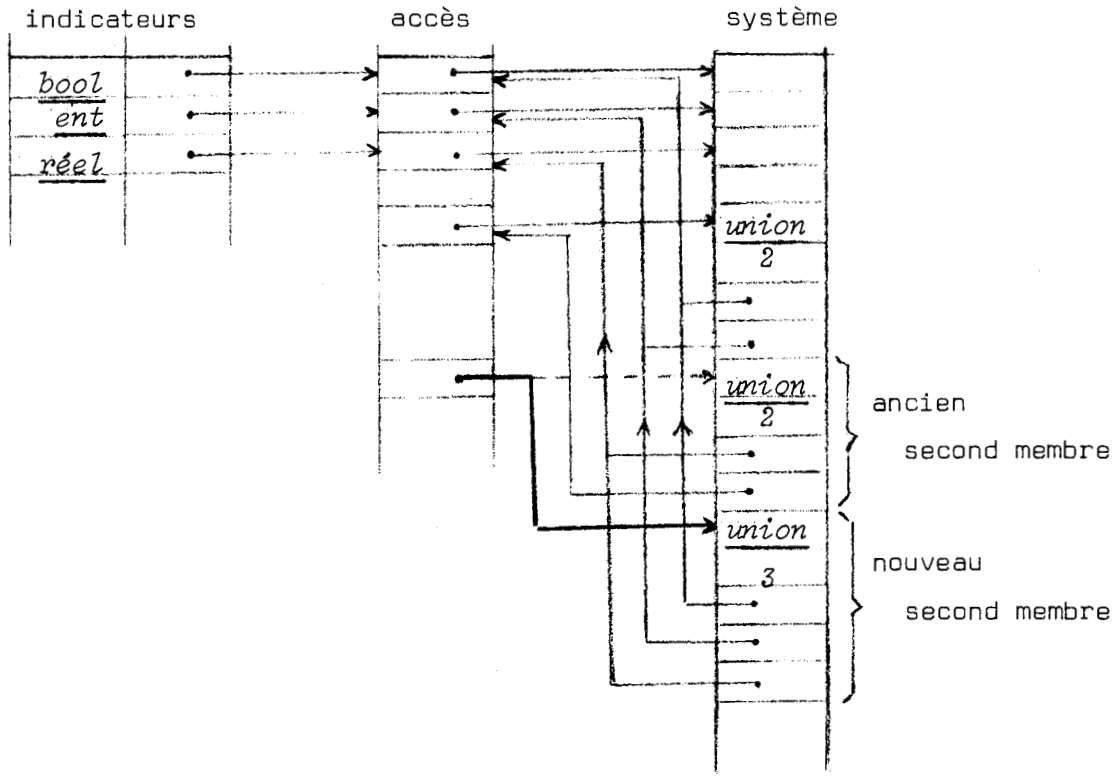
Le processus suivant est alors itéré jusqu'à ce que la pile soit vide :

Si l'inconnue qui se trouve au sommet de la pile définit un mode union, alors cette inconnue est dépilée, puis les inconnues constituant le second membre de l'équation qui lui est associée, sont placées dans la pile, enfin la variable booléenne prend la valeur vrai car il sera nécessaire d'écrire un nouveau second membre pour le mode union en cours de traitement ; dans le cas contraire, l'inconnue qui se trouve au sommet de la pile, après avoir été dépilée, est placée dans

le système après tous les seconds membres qui y figurent déjà, ceci afin de préparer l'éventuelle création d'un nouveau second membre.

Lorsque la pile est vide, et si la variable booléenne a la valeur vrai, le nouveau second membre, qui est déjà partiellement constitué, est complété et l'accès associé à l'inconnue représentant le mode union qui fait l'objet d'un développement est remplacé par l'accès à ce nouveau second membre ; si la variable booléenne a la valeur faux, il n'y a pas lieu de développer l'union et rien n'est modifié dans la représentation du système d'équations. L'ancien second membre sera toujours conservé.

Exemple : dans le cas du déclareur *union (réel, union (ent, bool))* nous aurons la configuration suivante,




Le développement des unions peut être réalisé au cours de la construction du système d'équations, à chaque fois qu'est écrit un nouveau second membre commençant par *union*. Cependant ce traitement ne peut être que partiel dans le cas où les déclareurs d'union comportent des indicateurs de mode non encore

définis pour spécifier les modes des composantes de l'union.

On trouvera en annexe, sous forme d'une partie de programme Algol 68, une description moins informelle de l'algorithme de développement des unions.

### 1.4. Réduction du système

Après les vérifications et traitements précédents, la comparaison des modes définis par deux inconnues différentes peut être entreprise à partir de la représentation interne du système d'équations. L'algorithme qui permet de déterminer si deux inconnues représentent le même mode sera décrit au paragraphe 2. En utilisant cet algorithme, la réduction du système consiste à comparer deux à deux les modes représentés par toutes les inconnues du système et, en cas d'égalité, à ne conserver qu'un seul accès pour ces deux inconnues en remplaçant l'adresse du second membre définissant l'une par celle du second membre définissant l'autre.

De même que pour les paragraphes précédents, il y aura intérêt à réaliser une réduction au fur et à mesure de la construction du système, c'est-à-dire à ne créer une nouvelle équation (en pratique un nouveau second membre) que dans le cas où il n'existe pas d'inconnue représentant le même mode dans la partie du système déjà construite. A cause de l'utilisation éventuelle d'indicateurs de mode avant leur définition, ce qui entraîne d'ailleurs le non-développement éventuel de certaines unions, cette réduction ne peut être que partielle ; elle sera néanmoins très intéressante dans le cas des modes usuels.

Il est important de remarquer, qu'après réduction du système d'équations définissant les modes utilisés dans un programme, la simple comparaison de deux inconnues (c'est-à-dire en pratique la comparaison de deux entiers) permet d'affirmer qu'elles représentent ou non le même mode. En outre il est possible de définir une relation d'ordre sur l'ensemble des modes utilisés dans un même programme, ce qui permet d'ordonner les modes composant des unions ; cette particularité peut être utilisée pour rechercher l'une des variantes (R.8.2.4.1.c) de la modification unir (voir chapitre 5).

### 1.5. Vérification de la condition sur les modes unions

Par définition, les modes composant un mode union ne doivent pas être *apparentés* deux à deux. Deux modes sont apparentés s'ils sont tous les deux *fermement modifiés* à partir d'un même mode. Enfin un mode μ (dit mode a posteriori)

est fermement modifié à partir d'un mode μ' (dit mode a priori) si μ' peut être obtenu à partir de μ par composition de modifications de mode autorisées en contexte ferme.

Rappelons que ces modifications sont :

*dérepérer*, qui transforme le mode μ en celui, μ', d'un repère de valeur de mode μ, ce qui peut être schématisé par $\mu' = \underline{rep}\ \mu$,

*déprocédurer*, qui transforme le mode μ en celui, μ', d'une procédure sans paramètre et à résultat de mode μ, ce qui peut être schématisé par

$$\mu' = \underline{proc}\ \mu,$$

*procédurer*, qui transforme le mode μ d'une procédure sans paramètre en celui, μ', de son résultat, ce qui peut être schématisé par

$$\mu = \underline{proc}\ \mu',$$

*unir*, qui transforme le mode μ d'une union en celui, μ', d'une des composantes de cette union, ce qui peut être schématisé par

$$\mu = (\underline{union}\ (\ldots, \mu', \ldots).$$

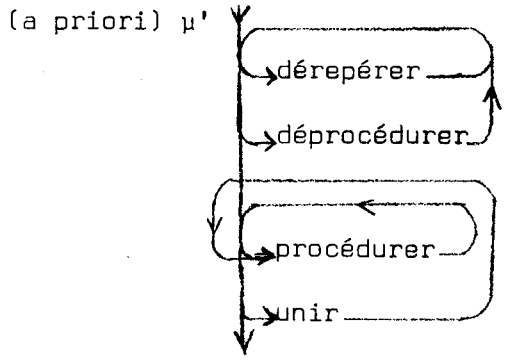
Remarquons qu'en toute rigueur unir permet aussi de transformer le mode

$$\mu = \underline{union}\ (\ldots, p, q, r, \ldots)$$

en $\mu' = \underline{union}\ (p, q, r)$.

Ce cas ne peut se produire pour le problème qui nous concerne, à savoir l'apparentement des modes composant un mode union, car ces derniers, par définition ne peuvent être à leur tour des modes unis ; d'ailleurs l'objet du paragraphe 1.3 a été de décrire un traitement (le développement des unions) qui permet de revenir à la forme stricte de la définition.

Ces modifications peuvent être composées entre elles, c'est-à-dire que le mode obtenu à la suite d'une modification peut à son tour subir une nouvelle modification. Le schéma suivant résume ces possibilités ; il n'est valable que dans le cas de la recherche de l'apparentement ; en effet il montre que pour unir il est nécessaire de procédurer au moins une fois ensuite.




μ' = *rep* μ et μ peut constituer un nouveau μ'

μ' = *proc* μ et μ peut constituer un nouveau μ'

μ = *proc* μ' et μ peut constituer un nouveau μ'

μ = *union* (..., μ', ...) et μ ne peut constituer un nouveau μ' que par procédurer.

Suppression des composantes égales

La représentation du système obtenue à l'issue de la phase de réduction peut contenir des seconds membres, commençant par *union*, où apparaissent des inconnues représentant le même mode puisque la syntaxe du langage autorise la nilpotence dans l'écriture des déclareurs d'union ; or un mode est apparenté à lui-même (il peut être obtenu à partir de lui-même par une suite vide de modifications autorisées en contexte ferme). Nous éliminerons donc, avant d'entreprendre la vérification même de la condition (C2), les composantes égales des modes union. Puisqu'il s'agit de déterminer si deux inconnues représentent le même mode, il y aura intérêt à le faire après réduction du système. Cette élimination peut être aisément réalisée en recopiant dans le second membre même la première inconnue si elle diffère des autres, puis la deuxième si elle diffère de celles qui restent et ainsi de suite. En outre, dans le cas où le nombre des inconnues constituant le nouveau second membre est égal à un, il y a erreur car un mode unir doit avoir par définition au moins deux composantes ; ainsi le déclareur *union (ent, ent)* est erroné bien que son écriture soit autorisée par la syntaxe du langage.

Le tri des composantes d'union peut être réalisé en même temps que la suppression de ces redondances (voir l'utilisation au chapitre 5).

Contrôle du non apparentement

Après suppression des composantes égales, la condition (C2) se vérifie aisément à partir de la représentation du système d'équation. Remarquons que si le second membre faisant l'objet de la vérification est de la forme *union* $x_1\ x_2 \ldots x_j \ldots x_k \ldots x_p$ (1) l'inconnue $x_k$ définissant elle-même un mode de la forme *proc*⁺ *union* $y_1 \ldots y_1 \ldots y_q$ (2), il faut et il suffit pour que $x_j$ et $x_k$ soient apparentés qu'il existe l, $1 \leq l \leq q$, tel que $x_j$ et $y_l$ le soient

aussi.

Dans une première phase nous rechercherons donc tous les modes qui peuvent être obtenus à partir de celui défini par (1), par une suite de modifications unir ou procédurer. Pour celà nous utiliserons un algorithme analogue à celui décrit pour le développement des unions.

Phase d'initialisation :

Les inconnues constituant le second membre faisant l'objet du contrôle sont placées dans une pile.

Le processus suivant est alors itéré jusqu'à ce que la pile soit vide :

L'inconnue se trouvant au sommet de la pile est dépilée puis considérée ; si l'inconnue considérée définit le mode d'une procédure sans paramètres, l'inconnue représentant le mode du résultat de cette procédure est considérée et ce test est réalisé de nouveau (cette phase revient à procédurer au maximum afin d'obtenir le mode représenté par l'inconnue dépilée) ;
si l'inconnue considérée définit un mode uni les inconnues du second membre associé sont placées dans la pile et l'étape est de nouveau exécutée (cette phase correspond à la modification unir) ;
dans tous les autres cas l'inconnue considérée est placée dans un tableau auxiliaire qui servira dans la deuxième phase.

Remarquons que cette première phase s'arrête en un nombre fini d'étapes lorsque la condition sur les déclarations de mode est vérifiée.

Au cours de la deuxième phase, toutes les inconnues du tableau auxiliaire sont comparées deux à deux :
si le mode représenté par l'une peut être obtenu à partir du mode représenté par l'autre au moyen d'une suite de modifications dérepérer ou déprocédurer, les inconnues définissent des modes apparentés et lès modes définis par le système ne vérifient donc pas la condition (C2).

Remarquons que cette erreur peut se manifester dans les composantes d'une union intervenant dans une autre union (par exemple $y_1$ et $y_q$ dans (2)). Dans un tel cas c'est indirectement que le mode union faisant l'objet de la vérification est erroné, car il utilise, pour sa définition, un mode union qui lui est erroné.

## 2 - TEST D'EGALITE DE DEUX MODES

Nous supposerons dans tout ce qui suit, que la représentation du système d'équations a été construite comme nous l'avons indiqué au chapitre 3. Les modes sont donc représentés par des entiers, à savoir les adresses, dans le vecteur ***accès***, des éléments contenant les adresses des seconds membres des équations. L'objet de ce paragraphe est la description d'un algorithme permettant de déterminer si deux tels entiers représentent le même mode.

### 2.1. Egalité de deux modes

Nous dirons que deux équations définissant des modes sont analogues si

- soit elles appartiennent toutes deux à $E_1 \cup E_2$ et les terminaux qu'elles contiennent sont identiques,
- soit elles appartiennent toutes deux à $E_3$, les terminaux qu'elles contiennent sont identiques et le nombre d'inconnues de leurs seconds membres est le même,
- soit elles appartiennent toutes deux à $E_4$, le nombre d'inconnues de leurs seconds membres est le même et les inconnues de même rang qui définissent des sélecteurs définissent les mêmes sélecteurs.

Associons alors au système d'équations un graphe dont :

- les sommets sont les paires d'inconnues $(x_i, x_j)$ du système d'équations,
- les arcs d'extrémité $(x_i, x_j)$ ont pour origine les paires formées par les inconnues de même rang dans les seconds membres des équations qui définissent $x_i$ et $x_j$ si ces équations sont analogues ; dans le cas contraire aucun arc n'admet la paire $(x_i, x_j)$ pour extrémité ; l'ensemble de ces sommets associés à des paires d'inconnues qui ne définissent pas des équations analogues est désigné par E. [3] a montré que : deux inconnues $x_i$ et $x_j$ représentent le même mode si et seulement s'il n'existe aucun chemin reliant un des sommets de l'ensemble E au sommet $(x_i, x_j)$.

Si nous voulons utiliser ce résultat pour construire ce graphe à partir de la représentation du système d'équations obtenue après développement des unions, nous devons tenir compte des deux autres particularités introduites par les équations qui définissent des modes unions, à savoir la commutativité et la nilpotence. Le problème de la commutativité peut être résolu en envisageant tous les graphes possibles correspondant à toutes les combinaisons possibles de toutes les inconnues figurant dans les seconds membres du système qui commencent

par *union*. Il semble plus simple de dire que deux seconds membres commençant par *union* définissent le même mode si l'ensemble des modes définis par les inconnues figurant dans l'un d'eux est égal à celui des modes définis par les inconnues figurant dans l'autre ; ce qui résoud aussi le problème de la nilpotence.

2.2. Description de l'algorithme pratique

Cet algorithme utilise directement les résultats du paragraphe précédent en n'explorant que la partie du graphe nécessaire à la comparaison des modes représentés par deux inconnues. De manière à faciliter la compréhension de la description qui va suivre nous rappelerons, avec la terminologie propre à ce chapitre, un résultat de [3].

Deux inconnues définissent des modes égaux si les seconds membres associés sont identiques ou bien, dans le cas où ils sont analogues, leurs inconnues de même rang définissent à leur tour des modes égaux. Pour les modes unions il faut et il suffit que pour chaque inconnue figurant dans l'un des seconds membres il existe dans l'autre second membre une inconnue définissant un mode égal et réciproquement.

L'algorithme utilise une pile principale *p1* et une pile auxiliaire *p2*. Les éléments de ces piles sont des paires d'inconnues, c'est-à-dire en pratique des doublets d'entiers. Dans la pile principale sont placés les doublets d'entiers dont il faut vérifier qu'ils représentent des modes égaux. Dans la pile auxiliaire sont placés les doublets d'entiers dont les équations associées sont analogues sans être identiques et sont supposées représenter des modes égaux. Au début *p1* et *p2* sont vides.

Phase d'initialisation :

Le doublet d'entiers représentant les inconnues dont les modes doivent être comparés est placé dans la pile principale *p1*.

Ensuite le processus suivant est itéré tant que la pile *p1* n'est pas vide (si la pile peut être vidée, les modes faisant l'objet de la comparaison sont égaux) :

Le doublet qui se trouve au sommet de la pile principale est dépilé et considéré. Ensuite

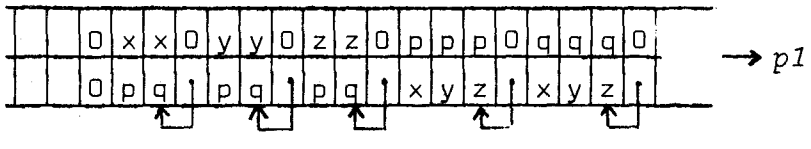
1. si *les deux entiers constituant le doublet considéré sont égaux*, ils repré-

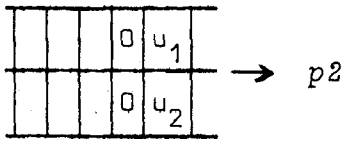
sentent le même mode et le processus est itéré ; dans le cas contraire,

2. si le *doublet considéré a déjà été rencontré*, c'est-à-dire qu'il se trouve déjà dans la pile auxiliaire *p2*, le processus est itéré ; dans le cas contraire,
3. si les *équations associées* aux entiers constituant le doublet considéré sont *analogues* et ne définissent pas des modes unions, les doublets formés par les entiers représentant les inconnues de même rang dans les seconds membres de ces équations sont empilés dans la pile principale *p1*, le doublet considéré est empilé dans la pile auxiliaire *p2* et le processus est itéré ; dans le cas contraire,
4. si les *équations associées* aux entiers constituant le doublet considéré ont des seconds membres qui commencent *tous deux par* union,
   a) un séparateur, dit *séparateur d'union*, constitué par deux entiers égaux à zéro, est empilé dans la pile principale *p1* et dans la pile auxiliaire *p2*,
   b) ensuite une série de *blocs de doublets* correspondant à toutes les combinaisons d'inconnues figurant dans l'un et l'autre seconds membres de ces équations est empilée dans *p1*, ces blocs étant séparés par des *séparateurs de blocs* constitués d'un entier égal à zéro et d'un entier pointant vers le doublet du bloc qui fera l'objet d'un traitement dans les étapes ultérieures (ce pointeur est initialisé à la valeur du pointeur de la pile *p1* juste avant mise en pile du séparateur qui le contient),
   c) le doublet considéré est placé dans la pile auxiliaire *p2*, c'est-à-dire que les entiers le constituant sont supposés représenter des modes égaux,

Exemple :

soit $(u_1, u_2)$ le doublet considéré et $u_1$ union x y z et $u_2$ union p q les équations associées, l'état des piles sera dans ce cas le suivant (après c)) :




(Remarquons qu'il y aura toujours au moins quatre blocs puisque les inconnues composant un second membre commençant par union sont en nombre supérieur

ou égal à deux). Pour que les inconnues dont les seconds membres commencent par *union*, définissent des modes égaux, il faut et il suffit que dans chacun des blocs ainsi constitués il existe au moins un doublet formé de deux entiers représentant le même mode. Les doublets constituant un bloc seront donc examinés les uns après les autres, pour celà,

d) le doublet qui se trouve au sommet du dernier bloc empilé dans la pile *p1* est empilé dans cette même pile, ce qui initialise l'examen des différents doublets de ce bloc, et un séparateur de bloc est placé dans la pile *p2* de façon à pouvoir aisément supprimer par dépilage de *p2*, les doublets correspondant à des hypothèses erronées (voir traitement des équations non analogues b) et d)), enfin

e) le processus est itéré,

5. si le doublet considéré est un *séparateur de blocs*,

a) le bloc qui se trouve au sommet de la pile principale *p1* est dépilé (en effet, avec les hypothèses faites dans *p2*, l'un au moins des doublets constituant ce bloc est formé d'entiers représentant le même mode et il est inutile de poursuivre l'examen des autres doublets constituant ce bloc, puis

b) le traitement du bloc qui se trouve alors au sommet de la pile *p1* est initialisé en empilant le premier doublet de ce bloc dans *p1* et un séparateur de bloc dans la pile auxiliaire *p2*, enfin

c) le processus est itéré,

6. si le doublet considéré est un *séparateur d'union*, le processus est itéré car les deux entiers constituant le doublet considéré sont égaux (voir 1. et 4. a)),

7. dans les autres cas, les *équations associées* aux deux entiers qui forment le doublet considéré *ne sont pas analogues* et ces entiers ne représentent donc pas le même mode ; plusieurs situations peuvent se présenter, ce qui donne lieu au

Traitement du cas où les équations ne sont pas analogues

a) s'il n'existe pas de séparateur de bloc dans la pile principale *p1*, il n'y a pas de doublet correspondant à deux modes unions en cours de traitement, il n'y a donc plus aucune alternative et les modes représentés par les entiers constituant le doublet initial sont différents ; autrement dit il existe un chemin reliant un sommet de l'ensemble E au sommet constitué des deux inconnues faisant l'objet de la comparaison et l'algorithme est achevé ; dans le cas

contraire, c'est-à-dire

b) s'il existe encore au moins un séparateur de bloc dans la pile *p1*, les éléments qui se trouvent au sommet de la pile *p1* et ceux qui se trouvent au sommet de la pile *p2* sont dépilés jusqu'au *premier séparateur de blocs exclu* (en effet les doublets de *p2* ainsi supprimés ont été supposés représenter des modes égaux alors que ceci n'est pas nécessairement vrai) ; ensuite,

c) s'il existe encore au moins un doublet à examiner dans le bloc qui se trouve au sommet de la pile principale *p1* (le pointeur constituant le séparateur de bloc qui est au sommet de *p1* est utilisé à cette fin), l'examen de ce doublet est initialisé en le plaçant au sommet de la pile *p1* et en exécutant de nouveau le processus ; dans le cas contraire, c'est-à-dire

d) s'il n'y a plus de doublet à examiner dans le bloc qui se trouve au sommet de *p1* (le doublet d'entiers définissant des modes union, qui est en cours de traitement définit deux modes différents), les éléments qui se trouvent au sommet de la pile *p1* ainsi que ceux qui se trouvent au sommet de la pile *p2* sont dépilés jusqu'au *premier séparateur d'union inclus* (pour les mêmes raisons qu'en b) en ce qui concerne *p2*) puis le traitement du cas où les équations ne sont pas analogues est de nouveau exécuté.

## 2.4. Remarques

A la fin de l'algorithme et dans le cas où les deux entiers formant le doublet initial représentent des modes égaux, tous les doublets se trouvant dans la pile auxiliaire *p2*, à l'exception des séparateurs d'unions et de blocs, sont constitués par des entiers représentant des modes égaux (la pile *p2* contient d'ailleurs le doublet initial). Cette remarque est utile dans le cas de la réduction du système d'équations, car non seulement un seul second membre sera conservé pour les deux inconnues représentées par les entiers du doublet initial, mais de plus un seul second membre sera conservé pour tous les autres doublets de la pile *p2*.

D'autre part, toujours dans le cas de la réduction du système, si la substitution d'inconnue est réalisée chaque fois que l'algorithme conclut à l'égalité des modes, le cas 1 du processus sera de plus en plus souvent réalisé, ce qui accélèrera l'algorithme. On trouvera en annexe une description plus concise de cet algorithme.

## 3 - REPRESENTATIONS CANONIQUES DES MODES

A l'issue de la phase de réduction du système d'équations, on obtient pour un programme donné, c'est-à-dire pour un système fini d'équations définissant des modes, une représentation canonique des modes utilisés dans ce programme, à savoir les entiers représentant les adresses des seconds membres des équations dans la représentation proposée au chapitre 3.

L'objet de ce paragraphe est de déterminer une *représentation canonique de tous les modes* telle que, de la même manière, la comparaison directe permette de conclure à l'égalité des modes représentés. Dans une première étape nous associerons une pseudo-arborescence à un mode, puis nous en déduirons deux représentations canoniques : l'une d'elle consiste à associer à chaque mode un entier positif et l'autre à représenter chaque mode par un mot de longueur finie.

### 3.1. Pseudo-arborescence associée à un mode

Donnons d'abord quelques définitions.

Système propre à un mode m :

C'est un *quadruplet* $S_m = (W, T, E, x_m)$

où $x_m \in W$ et m est le mode représenté par $x_m$ dans le système d'équations $S = (W, T, E)$.

Système réduit propre à un mode m :

Le système $S_m$ propre à un mode m est réduit si et seulement si

a) $\forall x_i, \forall x_j \in W : x_i \neq x_j \Rightarrow \mu(x_i) \neq \mu(x_j)$

en notant, pour tout $x \in W$, $\mu(x)$ le mode représenté par l'inconnue x ;

b) $\forall x_k \in W$ : $x_m$ dépend de $x_k$ (au sens des c-grammaires) ;

c) $\forall e_1 \in E$ telle que $e_1 = x_1$ *union* $x_{l_1}\ x_{l_2} \ldots x_{l_r}$, $x_{l_i} \neq x_{l_j} \Leftrightarrow l_i \neq l_j$ $(1 \leq i, j \leq r)$.

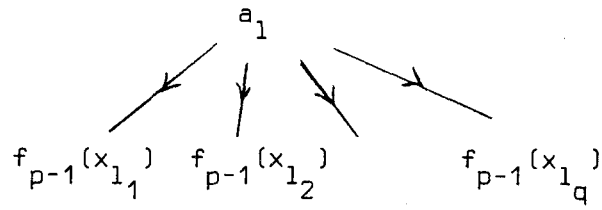
Pseudo-arborescence associée au système réduit propre à un mode :

Soit n la cardinalité de W.

Pour tout élément de E de la forme $x_1\ a_1\ x_{l_1}\ x_{l_2} \ldots x_{l_q}$,

définissons $f_o(x_1) = a_1$

et pour $1 \leq p \leq n$ $f_p(x_1)$ représente la pseudo-arborescence




Nous poserons $\phi(S_m) = f_n(x_m)$. Remarquons que $\forall p$, $0 \leq p \leq n$, $f_p(x_1) = a_1$ si l'équation a la forme $x_1\ a_1$.

$\phi(S_m)$ est une pseudo-arborescence sur T que nous dirons associée au système $S_m$.

Proposition 3.1 : quels que soient les systèmes réduits $S_m$ et $S'_m$ propres aux modes m et m',

$$\boxed{\phi(S_m) = \phi(S'_m) \Longrightarrow \mu(x_m) = \mu(x'_m)}$$

En effet nous constatons aisément qu'en considérant la représentation du système d'équations $S = (W \cup W', T, E \cup E')$ et en appliquant l'algorithme décrit au paragraphe 2.2 aux entiers représentant les inconnues $x_m$ et $x'_m$, ces inconnues représentent des modes égaux.

Remarque : La réciproque n'est pas vraie à cause de la commutativité des composantes d'union. Ainsi les deux pseudo-arborescences suivantes peuvent être associées au même mode et elles sont différentes :


## 3.2. Première représentation canonique des modes

Nous avons d'abord cherché une relation d'ordre sur l'ensemble des modes ; en effet une telle relation permet d'ordonner les inconnues représentant les modes composant une union, et la pseudo-arborescence associée à un système dont ces inconnues ont été ordonnées peut constituer une représentation canonique des modes. La première idée qui vient à l'esprit est d'utiliser l'ordre lexicographique sur l'ensemble des modes, ordre induit par un ordre défini sur

l'ensemble des signes syntaxiques minuscules ; la possibilité d'avoir des modes récursifs pour lesquels la récursivité utilise une ou plusieurs équations définissant un mode uni ne nous a pas permis d'exploiter cette idée. Nous avons donc été conduit à exhiber une injection de l'ensemble des modes dans l'ensemble des entiers positifs, ce qui fournit directement une première représentation canonique.

Définition 3.2.1. : *nombre de Goedel* associé à une suite finie d'entiers.
Pour toute suite finie d'entiers positifs $\Sigma = \{a_1, a_2, \ldots, a_k\}$

le nombre de Goedel associé à cette suite est défini par

$$G(\Sigma) = \prod_{j=1}^{k} [Pr(j)]^{a_j}$$

où Pr(j) représente le $j^{ième}$ nombre premier.

Proposition 3.2.1. : quelles que soient les suites finies $\Sigma_1$ et $\Sigma_2$

$$\Sigma_1 \text{ et } \Sigma_2 \text{ identiques} \iff G(\Sigma_1) = G(\Sigma_2)$$

Définition 3.2.2. : *pseudo-nombre de Goedel* associé à une suite finie d'*entiers*.
Pour toute suite finie de l entiers positifs $\Sigma = \{b_1, b_2, \ldots, b_l\}$

le pseudo-nombre de Goedel associé à cette suite est défini par

$$D(\Sigma) = 3^{l} \times 5^{max(\Sigma)} \times 7^{\nu}$$

où -max (Σ) représente le plus grand des éléments de la suite Σ, et
-ν est obtenu en dénombrant, dans un ordre donné, tous les *ensembles* de l entiers positifs inférieurs ou égaux à max(Σ); ces ensembles sont construits dans un ordre donné et à chacun est associé l'entier naturel qui est son numéro d'ordre ; ν est le numéro d'ordre associé à l'ensemble de l entiers identiques à celui des l entiers qui constituent la suite Σ. La valeur de ν est par exemple, celle de la proposition fermée :

*(ent numéro d'ordre := 0 ;*
*pour i1 jusqu'à max (sigma) faire*

*pour i2 jusqu'à i1 faire*
*co .... co*
*pour il jusqu'à il1 co représente la valeur courante de l'avant dernier indice co faire*
*si n plus 1 ;*
*co l'ensemble co sigma est identique à*
*co l'ensemble co (i1, i2, ..., il)*
*alors stop fsi ;*
*stop : numéro d'ordre)*

Proposition 3.2.2. : quelques soient les suites finies $\Sigma_1$ et $\Sigma_2$

$\Sigma_1$ et $\Sigma_2$ identiques à l'ordre près de leurs éléments
$\iff D(\Sigma_1) = D(\Sigma_2)$

Ceci résulte directement de la définition des pseudo-nombres de Goedel.

Il nous reste à utiliser ces résultats pour définir un nombre entier associé à un mode ; nous utiliserons pour celà la pseudo-arborescence associée à un mode.

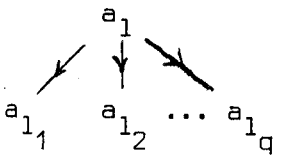
Définition 3.2.3. : *pseudo-nombre de Goedel* associé à un *mode*.
Nous associerons d'abord de façon biunivoque un nombre entier positif à chaque élément de l'ensemble T ; soit $\eta$ cette application. Le pseudo-nombre de Goedel est défini de façon récursive à partir de la pseudo-arborescence associé à un mode :

- appelons distance d'un noeud de la pseudo-arborescence à un autre la longueur du chemin qui les relie ;

- pour p = 0, 1, 2, ..., n, n étant le nombre d'éléments de W, à tout noeud $a_1$ dont la distance à la racine de la pseudo-arborescence est n-p, associons le nombre

$$\theta_p(a_1) = \eta(a_1) \text{ si aucun arc ne part du noeud } a_1 (p=0 \text{ ou } a_1 \in T_1 \cup T_5)$$
$$= G(\{\eta(a_1), \theta_{p-1}(a_{1_1}), \theta_{p-2}(a_{1_2}), \ldots, \theta_{p-1}(a_{1_q})\})$$

si la sous-pseudo-arborescence de sommet $a_1$ a la forme




avec $a_1 \neq$ *union*

$$= 2^{n(a_1)} \times D(\{\theta_{p-1}(a_{1_1}), \theta_{p-1}(a_{1_2}), \ldots, \theta_{p-1}(a_{1_q})\})$$

si la sous-pseudo-arborescence de sommet $a_1$ a la même forme que ci-dessus, mais avec $a_1$ = *union*.

Nous appellerons *pseudo-nombre de Goedel associé à un mode* m le nombre

$$\mathcal{N}(m) = \theta_n(a_o)$$

$a_o$ étant la racine de la pseudo-arborescence associée au mode m.

Proposition 3.2.3. : quels que soient les modes m et m'

$$m \text{ identique à } m' \Longleftrightarrow \mathcal{N}(m) = \mathcal{N}(m')$$

Le système d'équation qui a permis de définir la pseudo-arborescence associée à un mode étant *réduit* les problèmes posés par l'associativité des unions et leur nilpotence sont éliminés. Celui posé par la commutativité des composantes d'union l'est par l'utilisation de pseudo-nombres de Goedel dans le cas des modes unions, ces nombres caractérisant une suite d'entiers positifs à l'ordre près. D'autre part deux pseudo-arborescences associées à deux modes égaux peuvent différer par l'ordre des sous-arbres ayant pour racine l'élément *union*.

Corrollaire : il existe une relation d'ordre sur l'ensemble des modes.

## 3.3. Deuxième représentation canonique

L'utilisation des nombres de Goedel pour définir une représentation canonique des modes implique la manipulation de nombres entiers assez grands ; nous avons donc été amenés à rechercher une représentation plus pratique.

Nous avons vu que la pseudo-arborescence associée à un mode constituerait en elle-même une représentation canonique si les composantes d'union étaient

ordonnées ; nous savons maintenant que celà est théoriquement possible, grâce au corrollaire précédent.

Cependant nous pouvons remarquer que, lors du calcul du pseudo-nombre de Goedel associé à un mode, il est possible de ne pas utiliser les pseudo-nombres de Goedel associés à une suite d'entiers dans le cas d'une union ; il suffit en effet dans ce cas d'utiliser le nombre $G(\{\eta(a_1), \theta_{p-1}(a_{1_1}), \theta_{p-1}(a_{1_2}), \ldots \theta_{p-1}(a_{1_q})\})$ pour lequel les entiers $\theta_{p-1}(a_{1_j})$, $1 \leq j \leq q$, ont été ordonnés.

La deuxième représentation canonique que nous proposons met en oeuvre cette idée en conservant les éléments de T, au lieu de leur associer des entiers, et en utilisant des expressions parenthésées au lieu de nombres de Goedel. Ainsi, la notion de distance et les notations étant les mêmes qu'au paragraphe précédent, pour p = 0, 1, 2, ..., n, à tout noeud $a_1$ dont la distance à la racine de la pseudo-arborescence est n-p, nous associons l'expression parenthésée

$\varepsilon_p(a_1) = a_1$ si aucun arc ne part du noeud $a_1$,

$= a_1(\varepsilon_{p-1}(a_{1_1}), \varepsilon_{p-1}(a_{1_2}), \ldots, \varepsilon_{p-1}(a_{1_q}))$

dans le cas contraire et après avoir ordonné suivant l'ordre lexicographique les expressions parenthésées $\varepsilon_{p-1}(a_{1_j})$, $1 \leq j \leq q$, quand $a_1$ = *union*.

L'expression $\varepsilon_n(a_o)$ constitue une seconde représentation canonique des modes. Par exemple l'expression représentant le mode spécifié par *h* dans

*mode h = struct (rep h h, rep f f), f = struct (rep h h, rep f f)*

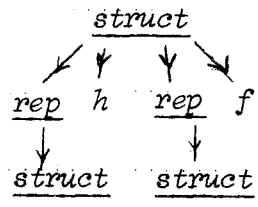
est *struct (rep (struct) h, rep (struct) f)*.

En effet les modes spécifiés par *h* et *f* sont les mêmes, les équations du système réduit propre au mode spécifié par *h* sont

$x_h$ *struct* $x_r$ *h* $x_r$ *f*

$x_r$ *rep* $x_h$

et la pseudo-arborescence associée est la suivante



3.4. Remarques

Tout mode pouvant être représenté par une expression parenthésée finie, il serait intéressant de construire la c-grammaire qui engendre le langage constitué par l'ensemble de ces expressions.

D'autre part les représentations canoniques précédentes supposant que les systèmes propres sont réduits, c'est-à-dire qu'elles utilisent le test d'égalité de mode. Il serait, aussi, intéressant de définir l'égalité des deux modes par l'égalité des expressions parenthésées associées ; ces expressions étant obtenues en prenant pour n le plus grand des nombres d'inconnues de chaque système propre aux deux modes, en supprimant l'une des deux expressions parenthésées identiques dans les unions et en ordonnant ces expressions comme ci-dessus.

# CHAPITRE 5

## IDENTIFICATION EN ALGOL 68.

## TRAITEMENT DES MODIFICATIONS.

La signification de certains objets externes, c'est-à-dire des unités syntaxiques comme les identificateurs, les opérateurs et les indicateurs de mode, est définie à l'aide de *déclarations* situées à des endroits donnés du programme ; cette définition peut ensuite être utilisée à un autre endroit de ce même programme. L'objet défini figure dans sa déclaration de définition, cette occurrence est appelée l'*occurrence de définition* de cet objet par opposition aux *occurrences d'utilisations* qui sont celles où l'objet est utilisé.

Algol 68 laisse la possibilité de donner plusieurs significations à un même objet, à condition, évidemment, de respecter certaines règles de bon sens destinées à éviter des ambiguités (quand un objet est utilisé il ne doit avoir qu'une signification possible). Il est par exemple possible d'écrire :

```
(réel x = 7.25, bool b ; lire (b) ;
   si b alors ent x = - 1 ; imprimer (x) fsi ;
   imprimer (x))
```

La première occurrence de l'identificateur $x$ est de définition, la deuxième l'est aussi ; la troisième est une utilisation de la deuxième et la quatrième est une utilisation de la première. C'est une des possibilités, offerte aussi dans une certaine mesure par d'autres langages, que d'associer à des identificateurs un certain domaine de validité (c'est-à-dire une certaine partie de programme).

Nous appellerons *identification* le processus qui permet d'associer à une occurrence d'utilisation d'un objet externe une et une seule occurrence de définition, donc de connaître la signification de cet objet. L'identification des identificateurs est assez simple. Celle des opérateurs est plus complexe car elle dépend du contexte dans lequel est utilisé un opérateur.

Le but du présent chapitre est de fournir des méthodes d'identification des différents objets externes nécessitant cette identification en Algol 68. Nous

décrirons plus particulièrement l'identification des opérateurs qui, comme nous le verrons, met en jeu les *modifications* de mode. Nous serons ensuite amenés à constater que l'identification des opérateurs peut être réalisée en même temps que le traitement des modifications dans leur ensemble et leur contrôle.

## 1 - IDENTIFICATION DES IDENTIFICATEURS, DES INDICATEURS

Pour les *identificateurs*, les méthodes classiques d'identification dans les textes à structure de blocs sont utilisables [7]. Il faut toutefois faire attention à ne pas considérer comme identificateurs les sélecteurs de champs qui figurent dans les déclareurs de modes structurés et les sélections. Les occurrences de définition des sélecteurs peuvent être reconnues lors de la constitution du système d'équation définissant les modes et les occurrences d'utilisation des sélecteurs sont toujours suivies du symbole *de* (il y a une exception dans le cas de la représentation du symbole *depuis* par *de* ; dans ce cas l'identificateur éventuel qui précède *de* est toujours précédé du symbole *pour*). D'autre part la reconnaissance des blocs, ou *régions* dans la terminologie Algol 68, nécessite un minimum d'analyse syntaxique ; en effet une notation de routine est une région dans laquelle figure des occurrences de définition d'identificateurs (paramètres formels) dont le domaine de validité s'étend à la notation de routine toute entière et non pas seulement aux paramètres formels La détermination du domaine de validité de l'identificateur de constante qui contrôle une instruction d'itération présente également quelques particularités.

En ce qui concerne les *indicateurs de mode* et les *indicateurs dyadiques*, leurs occurrences de définition sont de la forme :

... *mode m1 = ..., m2 = ...*
... *struct m3 = ( ), m4 = ...*
... *union m5 = ( ), m6 = ...*
... *priorité p1 = 4, p2 = 5, ...*

Nous considérons par ailleurs que les occurrences des *indicateurs monadique* qui figurent dans les déclarations d'opérations correspondantes sont des occurrences de définition :

... *op p3 = (m7a) m8 ..., p4 = (m9b) m10 ...*
... *op (m11) m12 p5 = ...*

Nous appellerons *fief* la partie propre d'une région, c'est-à-dire une région amputée de ses régions constituantes. La recherche des occurrences d'utilisation des indicateurs présente quelques difficultés ; en effet considérons les parties de programme suivantes :

(1) ... ; *proc a x ; éti* : ...
(2) ... ; *b y ; étiq* : ...

Dans le premier cas, *a* peut être a priori, soit un indicateur de mode, et nous sommes en présence d'une déclaration d'identité contracté, soit un indicateur dyadique, et nous sommes alors en présence d'une formule. Dans le deuxième cas, *b* peut être a priori, soit un indicateur de mode, et nous sommes en présence d'une déclaration d'identité contractée, soit un indicateur monadique, et nous sommes alors en présence d'une formule. La condition d'unicité sur la double définition dans un même fief d'un indicateur de mode ou d'un indicateur dyadique (R. 4. 4. 2. b) permet, pour identifier l'occurrence de définition de *a* dans le cas (1), de rechercher, dans les régions englobantes, soit une déclaration de mode, soit une déclaration de priorité définissant *a*. D'une manière analogue, la restriction imposée sur la création de nouveaux indicatifs (de mode) et de nouveaux indicatifs monadiques (R. 1. 1. 5. b), si nous admettons qu'elle s'applique localement (c'est-à-dire dans un même fief), permet, pour identifier *b*, de rechercher, dans les régions englobantes, soit une déclaration de mode, soit une déclaration d'opération d'opérateur monadique.

## 2 - IDENTIFICATION DES OPERATEURS

Dans le cas des opérateurs (monadiques ou dyadiques), l'identification est liée non seulement à la structure de blocs mais aussi au mode des opérandes de l'opérateur. En effet, un opérateur peut être *surchargé*, c'est-à-dire qu'il peut exister, pour un même indicateur, plusieurs définitions d'opérateurs avec des opérandes de modes différents (ceci peut se produire dans le même fief ou dans des fiefs différents). L'identification des opérateurs nécessite donc la comparaison des modes (voir chapitre 4).

Ainsi dans le domaine de validité des déclarations :

*op + = (ent a, b) ent : a - - b*
*op + = (réel p, q) réel : p - - q*

l'opérateur + de la formule *3 + 5* identifie la première occurrence de définition car dans cette formule les opérandes sont de mode entier, alors que l'opérateur + de la formule *3.14 + 7.25* identifie la deuxième occurrence de définition car, cette fois, les opérandes sont de mode réel.

Remarquons qu'il est aussi possible d'avoir dans le même fief une définition d'un opérateur monadique et d'un opérateur dyadique représenté par le même symbole ; par exemple :

*op truc = (ent a, b) : fant ; op truc = (ent c) : fant ;*

D'autre part les opérandes d'une formule sont en contexte ferme, ce qui autorise un certain nombre de possibilités de modifications (déprocédurer, dérepérer, unir, procédurer), des modes de ces opérandes.

Enfin les opérandes d'une formule peuvent être de nature plus complexe qu'un identificateur ou une notation ; ils peuvent être à leur tour des formules ce qui implique que le processus d'identification sera récursif et qu'il devra tenir compte de la *priorité* des opérateurs ; ce dernier point nécessite la différentiation des opérateurs monadiques et des opérateurs dyadiques lors de l'utilisation et de l'identification des indicateurs dyadiques (recherches de la signification "priorité" des opérateurs).

Par exemple dans la formule *a + - b * c / d*

nous identifierons d'abord le - monadique précédant l'identificateur *b*, ce qui permettra de déterminer le mode de la formule *-b*, nous pourrons donc identifier ensuite le * dyadique en fonction du mode de *-b* et de celui de *c* (dans le cas où la priorité de * est supérieure à celle de + dans le fief où figure cette formule), ce qui fournira le mode de *-b * c*, etc.

Nous verrons au paragraphe 3 que l'équilibrage des modes dans le cas où les opérandes ne sont pas des modandes, c'est-à-dire des propositions qui n'ont pas de mode propre, complique aussi le processus d'identification des opérateurs.

Le principe de la solution que nous proposons consiste à faire du *calcul de mode* après avoir transformé la structure d'opérateurs du programme par un passage à la notation postfixée. Cette transformation nécessite la mise en oeuvre de techniques classiques [4] qui utilisent la signification "priorité" des opérateurs. Les valeurs faisant l'objet du calcul sont les modes "possédés" par les identificateurs, notations et générateurs. En pratique, nous utiliserons

les entiers qui représentent ces modes dans le système d'équations que nous supposerons réduit (voir chapitre 4, paragraphe 1.4). Le mode d'un identificateur est celui du déclareur formel de la déclaration d'identité définissant cet identificateur (voir paragraphe 1) ; le mode d'une notation autre qu'une notation de routine est fourni par l'automate qui analyse ces notations (il s'agit toujours, dans ce cas, d'un mode primitif) ; le mode d'une notation de routine se déduit de celui de ses paramètres formels et de son forceur ; enfin celui d'un générateur se déduit directement du déclareur effectif. L'expression en notation postfixée est évaluée au moyen d'une pile de mémoires de travail et de façon classique.

Quand, en balayant de gauche à droite la chaîne postfixée, un opérande est rencontré, son mode est mis en pile ;

quand un opérateur monadique est rencontré, l'identification de l'indicateur monadique qu'est cet opérateur permet de déterminer une déclaration d'opération susceptible de définir cet opérateur monadique ; cette déclaration contient le mode *m1* de l'opérande formel ;

si le mode *m* qui se trouve au sommet de la pile de travail peut être *fermement modifié à partir du* mode *m1*, le mode *m* est remplacé par le mode *m2* du résultat de la routine possédé par l'opérateur de cette déclaration et l'identification de l'opérateur est terminée ;

dans le cas où le mode *m* n'est pas fermement modifié à partir de *m1*, la recherche d'une autre déclaration d'opération définissant l'indicateur monadique est poursuivie tout d'abord dans le même fief puis dans les fiefs englobants. Si cette recherche est infructueuse le programme est erroné.

Dans le cas d'un opérateur dyadique, après recherche d'une déclaration d'opération susceptible de définir cet opérateur dyadique, les deux modes qui se trouvent au sommet de pile sont utilisés pour déterminer si cette déclaration convient ; dans le cas où les modes a priori des opérandes effectifs sont fermement modifiés à partir des modes a posteriori des opérandes formels de même rang l'identification de l'opérateur dyadique est terminée et le mode du résultat, déduit de la déclaration, remplace les deux modes du sommet de pile.

Un certain nombre de symboles du langage doivent être considérés comme des opérateurs ne donnant pas lieu à identification, mais influant sur le processus d'identification par le mode de leur résultat. Ainsi pour un connecteur de conformité ou d'identité le résultat sera toujours de mode booléen quelque soit le mode

de ses opérandes. D'autre part l'opérateur de sélection *de* aura toujours comme opérandes un sélecteur et un mode de valeur structurée ; le secondaire d'une sélection étant en position ferme le mode de la valeur structurée pourra subir certaines modifications (voir paragraphe 6.4) ; il sera possible de contrôler l'existence du sélecteur et de déterminer le mode de la sélection. Nous considérerons qu'un appel ou une tranche sont représentés dans la chaîne postfixée par des opérateurs : l'*opérateur d'appel* et l'*opérateur de tranche* ; ces opérateurs donneront lieu aussi à des traitements particuliers ; dans le cas d'un appel le mode du résultat est évident et dans celui d'une tranche le nombre d'indices permettra de déduire son mode.

D'autres opérateurs généralisés doivent aussi être considérés car ils peuvent apparaître dans des opérandes ; il modifieront donc la pile des mémoires de travail ; par exemple, dans certains cas le ; devra "neutraliser" une formule, ce qui provoquera le dépilage d'un élément.

Ces remarques montrent que l'identification devra être menée conjointement avec un traitement complet des modifications, ce qui permettra une vérification de la correction de tous les modes utilisés. L'identification des opérateurs n'apparait alors que comme un cas particulier par rapport aux autres traitements concernant les modifications.

Nous allons détailler, dans les paragraphes suivants, les solutions que nous proposons pour résoudre les problèmes soulevés par l'existence d'opérandes qui ne sont pas des modandes (équilibrage en contexte ferme), pour déterminer l'existence des suites de modifications élémentaires en contexte ferme et en contexte fort. Puis nous détaillerons le traitement des opérateurs généralisés, nécessaire pour réaliser seulement l'identification des opérateurs. Enfin nous aborderons le contrôle des modes utilisés dans un programme et la détermination de toutes les modifications.

## 3 - EQUILIBRAGE EN CONTEXTE FERME

Les opérandes d'une formule peuvent être encore plus complexes que ceux que nous avons jusqu'alors envisagés ; il s'agit du cas où ce sont des propositions collatérales, des propositions conditionnelles (et leurs extensions) et des propositions fermées contenant une "vériatable" proposition sérielle (c'est-à-dire contenant au moins un acheveur). Toutes ces propositions peuvent être l'objet d'un *équilibrage* de modes.

Une proposition collatérale en contexte ferme permet de fournir une valeur multiple, elle ne permet jamais de fournir une valeur structurée. En outre un primaire d'appel ne peut pas être une proposition collatérale car son mode doit commencer par "procédure avec paramètre ...".

Une proposition collatérale peut se schématiser, en ce qui concerne les modes, par l'expression $(m_1, m_2, m_3, \ldots, m_n)$ $n \geq 2$
les $m_i$ étant les modes des propositions unitaires constituantes.

Dans les autres cas d'équilibrages, il s'agit d'un choix entre plusieurs calculs possibles. Il n'y a pas construction d'une valeur multiple, donc pas apparition d'un nouveau mode.

Une proposition fermée contenant au moins un acheveur ou une proposition conditionnelle (éventuellement sous forme d'extensions) peut se schématiser, en ce qui concerne les modes, par l'expression $\{m_1 \mid m_2 \mid m_3 \mid \ldots \mid m_p\}$ $p \geq 2$
les $m_i$ étant, de la même manière les mode des propositions constituantes. Il est important de distinguer les deux types d'équilibrage au moyen de deux sortes de parenthèses (ici les parenthèses ordinaires et les accolades).

L'équilibrage consiste à autoriser, pour des propositions collatérales ou fermées se trouvant en contexte ferme, que toutes les propositions constituantes puissent être en contexte fort sauf une qui doit être en contexte ferme, ce qui peut se schématiser par

( fort, ... , fort , ferme , fort , ... , fort )

et par

{ fort| ... | fort | ferme | fort | ... | fort }

La définition des propositions étant récursive, une proposition collatérale ou fermée peut à son tour contenir des propositions collatérales ou fermées ; ces dernières peuvent donc se trouver en contexte fort, ce qui peut conduire à avoir comme opérandes des propositions qui peuvent être schématisées par :

( fort , {fort|fort|fort} , (fort, fort, fort) , {fort|ferme} )

Dès lors des valeurs structurées fournies par des propositions collatérales peuvent se trouver à l'intérieur de propositions collatérales ou fermées constituant l'opérande d'un opérateur.

Exemple : *op r = ([ ] compl a) réel : ... ; compl z = p i q ; ...*
*r (1, si b1 alors pi sinon 7 fsi, (0,1),*
*si b2 alors pi / 2 sinon z fsi)*

Dans le cas d'un opérateur monadique, la solution déjà décrite au paragraphe 2 devra être complétée comme suit :

La proposition fermée donnant lieu à équilibrage se trouve dans la pile de travail sous forme de suites de modes séparés par des "," ou par des "|" et encadrés par des parenthèses ou des accolades ; en effet les propositions ne donnant pas lieu à équilibrage ont déjà été traitées. Le mode de l'opérande formel est considéré. La proposition est alors explorée, par exemple *de droite à gauche*, sans détruire le contenu de la pile car il peut y avoir plusieurs essais à tenter. Au cours du balayage de l'expression chaque élément donne lieu aux traitements suivants :

- une parenthèse fermante provoque l'incrémentation d'un compteur de parenthèses initialement à zéro, puis, si le mode considéré est de la forme "rang de μ", le mode μ est alors considéré à sa place et l'élément suivant est examiné, sinon l'exploration est reprise à la première parenthèse ouvrante correspondant à la parenthèse fermante en cours de traitement ;

- une parenthèse ouvrante provoque la décrémentation du compteur de parenthèses ; si ce dernier atteint la valeur zéro et si aucun des modes examinés n'a pu être fermement modifié à partir du mode de l'opérande formel, il n'existe pas de modification ; sinon le mode de la valeur multiple dont les éléments sont du mode considéré est considéré à sa place et l'élément suivant est examiné ;

- une accolade fermante provoque l'incrémentation du compteur de parenthèses, puis l'élément suivant est examiné ;

- une accolade ouvrante provoque la décrémentation du compteur de parenthèses ; si ce dernier atteint la valeur zéro, les mêmes traitements que pour une parenthèse ouvrante sont réalisés ; sinon l'élément suivant est examiné (sans changer le mode considéré) ;

- les séparateurs tels que la virgule ou la barre verticale sont ignorés ;

- dans les autres cas l'élément examiné est un mode ; si le mode examiné est *fermement modifié à partir* du mode considéré, l'identification, pour la partie de la proposition déjà traitée est possible, il reste à vérifier que les modes non encore examinés peuvent être fortement modifiés à partir du mode considéré ce qui peut être réalisé par une exploration analogue à celle déjà décrite ;

dans le cas contraire, si le mode examiné est *fortement modifié* à partir du mode considéré l'élément suivant est examiné, sinon l'exploration est reprise à la première parenthèse ouvrante obtenue en sautant les paires de parenthèses ou d'accolades associées.

Exemple : considérons les déclarations suivantes

(1) *op + = (réel a) ...*
(2) *op + = (ent a) ...*

Pour la formule monadique *+ si b alors 3.14 sinon 4 fsi* l'expression qui se trouve dans la pile a la forme {*réel, entier*}. Dans ce cas, si la déclaration (2) est la première testée et si l'exploration se fait de droite à gauche, la vérification mentionnée ci-dessus est nécessaire.

Dans le cas d'un opérateur dyadique, cette méthode est mise en oeuvre deux fois si les deux opérandes doivent faire l'objet d'un équilibrage ; les deux propositions collatérales qui se trouvent au sommet de la pile sont alors utilisées. La méthode décrite fonctionne aussi dans le cas où l'équilibrage se réduit à un simple choix entre plusieurs modes possibles (l'opérande est alors une proposition conditionnelle comme dans l'exemple ci-dessus).

Naturellement l'opérateur identifié remplace dans la chaîne postfixée l'indicateur adique représentant l'opérateur avant identification.

## - COMPOSITION DES MODIFICATIONS EN CONTEXTE FERME

Nous avons vu qu'il fallait déterminer si un mode est ou non *fermement modifié à partir d'un autre mode*. Nous avons déjà rappelé (chapitre 4, paragraphe 1.5) quelles étaient les modifications autorisées en contexte ferme et comment il était possible de les composer. L'algorithme décrit ci-après utilise presque directement la définition des modifications et de leurs règles de composition. Il s'en éloigne un peu par le fait que le mode a priori est systématiquement procédure, quitte à rectifier, par la suite, les erreurs éventuellement commises en procédant ainsi (un procédurage et un déprocédurage ne peuvent se suivre). La seule complication vient de la modification unir qui oblige, dans certains cas, à empiler les modes des composantes d'union pour les tester une par une. D'autre part, nous supposerons que ces composantes ont fait l'objet d'un tri (voir chapitre 4, paragraphe 1.4), ce qui permet de simplifier la détermination de la modification unir décrite par la règle R 3.2.4.1. d.

Nous avons d'ailleurs distingué pour *unir* le cas où la modification est décrite par la règle R 3.2.4.1. b (*unir 1*) de celui où elle l'est par la règle R.3.2.4.1. d (*unir 2*). Dans le premier cas le mode a priori est un mode non uni et le mode a posteriori un mode uni ; dans le second les modes a priori et a posteriori sont tous deux des modes unis. Dans le premier cas il sera utile de conserver le mode de la valeur qui doit être uni (ce mode sera nécessaire à l'exécution pour les éventuels tests de conformité).

S'il existe une suite de modifications élémentaires permettant de passer du mode a priori au mode a posteriori l'algorithme fournit dans un tableau (*modif*) cette suite de modifications ; le contenu de ce tableau sera utilisé lors du traitement des modifications dans leur ensemble pour insérer les opérateurs implicites correspondants. Nous avons considéré en fait que procédurer et unir 1 modifient le mode a posteriori, alors que dérepérer, déprocédurer et unir 2 modifient le mode a priori ; ceci simplifie la recherche des modifications mais entraîne la nécessité de rétablir l'ordre de composition pour obtenir un ordre correct.

L'algorithme, qui utilise une pile peut être décrit, schématiquement comme suit :

Test d'égalité :

Si le mode *a priori* et le mode *a posteriori* sont *égaux*, l'algorithme est *terminé* et il existe une suite de modifications, éventuellement vide, permettant de passer de l'un à l'autre. Le test d'égalité se réduit à une simple comparaison entre deux entiers dans le cas où les modes sont définis par un système d'équations réduit.

Dans le cas où les modes a priori et a posteriori sont différents,

- si le mode *a posteriori* est un mode *uni* :

. si le mode *a priori* est aussi un mode *uni*, le mode a priori initial est fermement modifié à partir du mode a posteriori initial si l'ensemble des modes des composantes du premier est inclus dans celui des composantes du deuxième ; il s'agit alors de la modification *unir 2* et l'algorithme est *terminé* si cette inclusion est fausse le mode a posteriori (qui est uni) est décomposé (voir ci-dessous) ;

. si le mode *a priori* n'est *pas* un mode *uni*, il est *dérepéré* et *déprocédurê* au maximum ; au cours de ce traitement, à chaque fois qu'un déprocédurage

ou un dérepérage élémentaire peut être réalisé, le test d'égalité est à nouveau exécuté ; quand le dérepérage ou déprocédurage n'est pas possible le mode a posteriori est décomposé ;

- si le mode *a posteriori* est celui d'une *procédure sans paramètres*, il est remplacé par le mode du résultat de cette procédure (c'est-à-dire que le mode a priori est systématiquement *procédure*), puis le test d'égalité est de nouveau exécuté ;

- dans le cas contraire le mode a priori est *dérepéré* et *déprocédure* au maximum comme ci-dessus, mais quand le dérepérage ou le déprocédurage n'est plus possible l'alternative suivante (provenant de la décomposition d'un mode union) est examinée (voir ci-dessous) ;

Décomposition d'un mode union :

Cette décomposition correspond à la modification *unir 1*. Elle consiste à empiler un entier spécial, dit séparateur de blocs d'union, puis les composantes du mode a posteriori. Ensuite l'alternative suivante, c'est-à-dire le mode qui se trouve au sommet de la pile, est examinée.

Examen de l'alternative suivante :

Si, lorsque cette séquence doit être exécutée, *la pile est vide*, il n'y a plus d'alternative pour unir 1 ; l'algorithme est donc terminé et le mode a priori initial *ne peut être modifié* à partir du mode a posteriori initial ; dans le cas contraire si le sommet de pile est un séparateur de blocs d'union, il est dépilé puis, si la pile n'est pas vide (sinon les conclusions seraient les mêmes que ci-dessus), toutes les modifications élémentaires procédurer, mémorisées dans le tableau depuis la modification unir 1 correspondant à ce séparateur, sont supprimées du tableau ainsi que cette modification unir 1, après quoi la même séquence que dans le cas où le sommet de pile n'est pas un tel séparateur est exécutée ; c'est-à-dire que le sommet de pile, après avoir été dépilé, remplace le mode a posteriori, le mode a priori initial remplace le mode a priori courant, toutes les modifications concernant le mode a priori (déprocédurer et dérepérer) sont supprimées du tableau, enfin le test d'égalité est à nouveau exécuté.

On trouvera en annexe une description plus concise de cet algorithme sous la forme de la déclaration d'un opérateur à opérandes entiers et à résultat booléen ; le résultat est vrai si l'entier constituant l'opérande gauche représente un mode fermement modifié à partir du mode représenté par l'entier constituant l'opérande droit, et faux dans le cas contraire.

## 5 - COMPOSITION DES MODIFICATIONS EN CONTEXTE FORT

Le traitement de l'équilibrage des modes implique l'utilisation d'un algorithme permettant de déterminer si un mode est ou non *fortement modifié à partir d'un autre*. Les modifications élémentaires autorisées en contexte fort sont, outre celles autorisées en contexte ferme, les quatre suivantes : élargir, mettre en rang, hisser et neutraliser.

*neutraliser* transforme, dans certaines conditions, le genre neutre lorsque ce dernier est requis par le contexte en n'importe quel mode a priori ; l'opérande d'un opérateur ne pouvant être de genre neutre, cette modification ne sera pas prise en compte pour l'identification dans le cas de l'équilibrage

*hisser* permet de fournir un mode à des bases qui n'ont pas de mode propre, à savoir les sauts (*allera étiquette*), le fantome *fant* et le nihil *nil* ; pour *nil* le mode a posteriori doit toujours commencer par "repère de" ; pour *fant* le mode a posteriori peut être quelconque, mais nous distinguerons le cas où ce dernier commence par "union" du cas où il n'en est pas ainsi (voir R.3.2.7.2.a) ; enfin pour les sauts, le mode posteriori peut aussi être quelconque, mais nous distinguerons cette fois le cas où c'est le mode d'une procédure sans paramètre du cas où il n'en est pas ainsi (voir R.3.2.7.2.b) ;

*mettre en rang* transforme le mode a posteriori μ d'une valeur multiple soit en une base qui est vide, soit en un mode a priori μ' qui est celui des éléments de la valeur multiple, ce qui peut être schématisé par

μ = [ ] μ' ;

cette modification transforme aussi le mode a posteriori *rep* μ d'un repère de valeur multiple en un mode μ' qui est celui des éléments de cette valeur multiple, ce qui peut être schématisé par

μ = *rep* [ ] μ" et

μ' = *rep* μ" ;

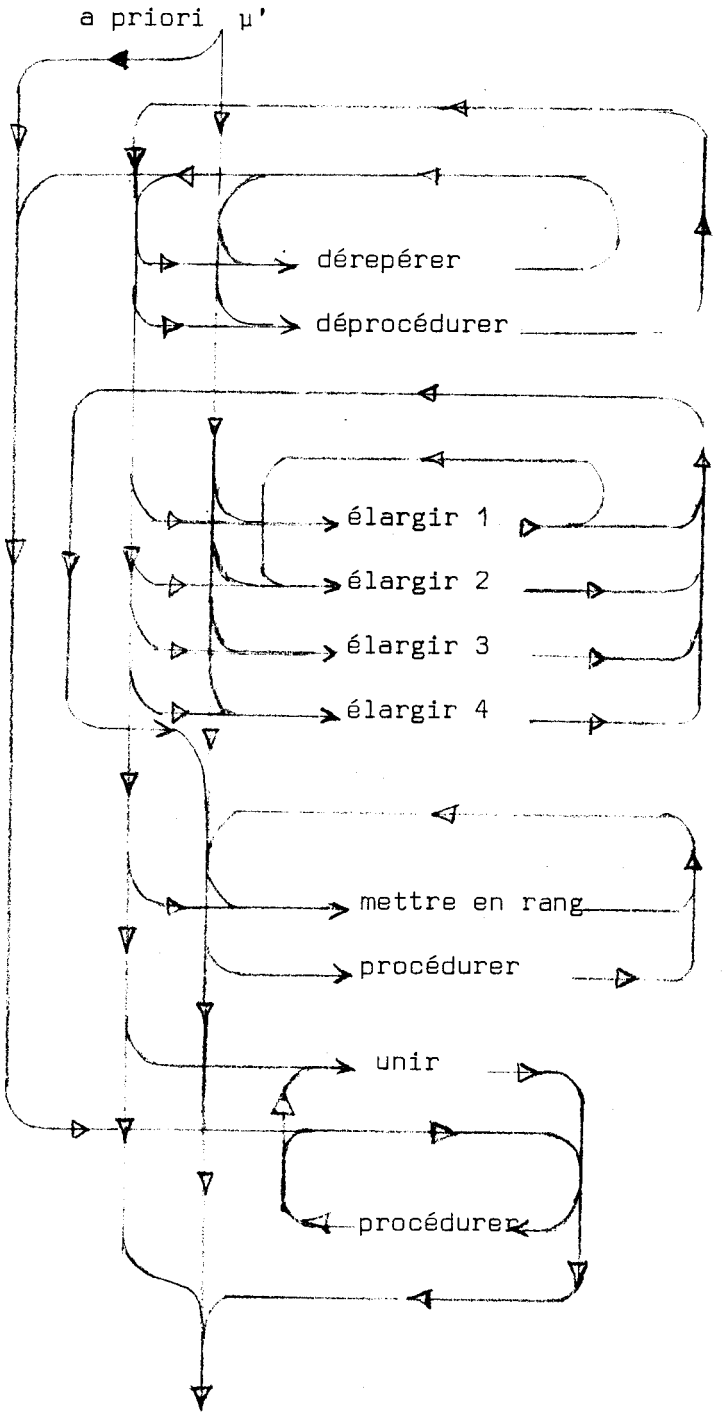
nous distinguerons donc ces trois cas ;

*élargir* transforme le mode a posteriori μ en un mode a priori μ' suivant l'un des quatre schémas suivants (il existe des schémas analogues pour les versions plus longues) :

1) μ = *réel* μ' = *ent* 2) μ = *compl* μ' = *réel*

3) μ = [ ] *bool* μ' = *bits* 4) μ = [ ] *car* μ' = *cat*

Ces modifications peuvent être composées entre elles comme nous l'avons vu au chapitre 4 ; cependant, en contexte fort, avant unir 1 seules les quatre modifications autorisées en contexte ferme peuvent être utilisées ; par ailleurs le hissage a toujours lieu. par définition en une seule fois, il ne peut donc être composé avec aucune autre modification élémentaire. Le schéma ci-dessous résume les possibilités de composition de modifications en contexte fort pour le seul cas du traitement de l'équilibrage en contexte ferme.




$\mu' = \underline{rep}\ \mu$

$\mu' = \underline{proc}\ \mu$

$\mu' = \underline{ent}$ et $\mu = \underline{réel}$

$\mu' = \underline{réel}$ et $\mu = \underline{compl}$

$\mu' = \underline{bits}$ et $\mu = [\ ]\ \underline{bool}$

$\mu' = \underline{cat}$ et $\mu = [\ ]\ \underline{car}$

ou $\mu'$ = vide et $\mu = [,,]\ \mu''$

ou $\mu = [\ ]\ \mu'$

ou $\mu = \underline{rep}\ [\ ]\ \mu''$ et $\mu' = \underline{rep}\ \mu''$

$\mu = \underline{proc}\ \mu'$

ou $\mu = \underline{union}\ (\ldots, \mu', \ldots)$

ou $\mu = \underline{union}\ (\ldots, p, q, r, \ldots)$

et $\mu' = \underline{union}\ (p, q, r)$

$\mu = \underline{proc}\ \mu'$

L'algorithme proposé utilise en partie le précédent et il fournit aussi dans un tableau la suite de modifications élémentaires permettant de passer du mode a priori au mode a posteriori.

Nous supposerons que le "vide", *nil*, *fant* et les sauts sont caractérisés par des modes particuliers.

- Si le mode *a priori* est celui d'un *saut*, deux cas peuvent se présenter :

. si le mode a posteriori est celui d'une procédure sans paramètre, (il faudra transformer ce saut en routine), il s'agit de la modification *hisser 1* ;

. si le mode a posteriori n'est pas celui d'une procédure sans paramètres (le saut sera exécuté), il s'agit de la modification *hisser 2* ;

. dans les deux cas l'algorithme est terminé ;

- Si le mode *a priori* est celui d'un *fantome*, deux éventualités peuvent encore se présenter suivant que le mode a posteriori commence ou non par "union" ; il leur correspond deux modifications élémentaires différentes : *hisser 3* et *hisser 4* ; de toute manière l'algorithme est terminé (nous remarquerons que le genre a posteriori ne peut être neutre dans le cas d'un opérande) ;

- Si le mode *a priori* est celui de *nil*, alors

. dans le cas où le mode *a posteriori* est celui d'un *nom*, il s'agit d'un hissage (*hisser 5*),

. dans le cas contraire il n'existe pas de modification et l'algorithme est terminé.

- Si le mode *a priori* est celui du *vide*, alors

. si le mode *a posteriori* est celui d'une *valeur multiple*, il s'agit de la mise en rang du vide (*mettre en rang 3*) et l'algorithme est terminé,

. sinon le mode a priori est procédure, ce qui est possible si le mode a posteriori est celui d'une procédure sans paramètre ; le mode du résultat de cette procédure constitue le nouveau mode a posteriori et le test précédent est de nouveau réalisé ; dans le cas où le procédurage n'est pas possible, il n'existe pas de modification et l'algorithme est terminé.

Test d'égalité :

Si le mode *a priori* et le mode *a posteriori* sont *égaux*, l'algorithme est terminé et il existe une suite de modifications élémentaires, éventuellement vide, permettant de passer de l'un à l'autre.

Dans le cas où ces deux modes sont différents ;

- si le mode *a posteriori* est un mode *uni* et si le mode *a priori* est *fermement modifié* à partir du mode *a posteriori* alors il existe une suite de modifications élémentaires et l'algorithme est terminé ;

- si le mode *a posteriori* est celui d'une *procédure sans paramètres*, il est remplacé par le mode du résultat de cette procédure (procédurage systématique), puis le test d'égalité est de nouveau exécuté ;

- si le mode *a posteriori* est celui d'une *valeur multiple*, il est remplacé par le mode des éléments de cette valeur multiple (*mettre en rang 1*), puis le test d'égalité est de nouveau exécuté ;

- si le mode *a posteriori* est de la forme "repère de rang de *m2*", alors

. si le mode *a priori* est celui d'un *nom*, c'est-à-dire de la forme "repère de *m1*", et si *m1* est égal à *m2*, il s'agit de la mise en rang d'un nom (*mettre en rang 2*) et l'algorithme est terminé ; si *m1* est différent de *m2* et si ce dernier mode est de la forme "rang de *m3*", le test précédent est réalisé de nouveau avec *m3* à la place de *m2* ;

. si le mode a priori n'est pas celui d'un nom ou bien si m2 n'est pas le mode d'une valeur multiple, le mode *a priori* est *dérepéré* et *déprocéduré* et à chaque fois qu'une telle modification élémentaire peut être réalisée le test d'égalité est à nouveau réalisé ; quand ces modifications ne sont plus possibles, l'algorithme est terminé et le mode a priori initial n'est pas fortement modifié à partir du mode a posteriori initial ;

- si le mode *a posteriori* est celui des valeurs *complexes*, alors

. si le mode *a priori* est celui des valeurs *réelles*, il s'agit de la modification élargir (*élargir 2*) et l'algorithme est terminé, sinon

. si le mode *a priori* est celui des valeurs *entières*, il s'agit de la composition de deux élargissements (*élargir 1* et *élargir 2*) et l'algorithme est terminé ;

. dans le cas contraire le mode *a priori* est *dérepéré* et *déprocéduré* comme ci-dessus ;

- si le mode *a posteriori* est celui d'une valeur *réelle* (resp. d'un *tableau de booléens, de caractères*) et si le mode a priori est celui d'une valeur entière (resp. d'une valeur de mode *bits*, *catène*), il s'agit de la modification *élargir 1* (resp. *élargir 3*, *élargir 4*) et l'algorithme est terminé ; dans le cas

contraire le mode a priori est dérepéré et déprocéduré comme ci-dessus.

On trouvera aussi en annexe une description plus concise de cet algorithme sous forme d'une déclaration d'opération.

## 6 - TRAITEMENT DE CERTAINS OPERATEURS GENERALISES POUR LE PROCESSUS D'IDENTIFICATION

Les opérateurs généralisés (confrontations, sélections, appels, etc.) n'ont pas besoin d'être identifiés, en tant qu'indicateurs ou en tant qu'opérateurs, car les symboles qui les représentent ne peuvent être utilisés que dans une seule situation bien précise. Cependant ces opérateurs doivent se comporter comme les autres opérateurs pour le calcul de mode ; l'objet de ce paragraphe est de déterminer les traitements nécessaires pour réaliser le processus d'identification.

L'examen de la syntaxe permet de fixer des priorités pour ces opérateurs. Nous allons examiner chaque cas l'un après l'autre.

### 6.1. Connecteurs de conformité et d'identité

Le résultat des confrontations contenant cet opérateur est toujours de mode booléen ; les opérateurs généralisés *::*, *::=*, *:=:* et *:≠:* seront donc considérés comme des opérateurs dyadiques possédant une routine à résultat booléen. Les modes des opérandes sont ignorés ; les deux opérandes qui se trouvent au sommet de la pile seront remplacés par l'entier représentant le mode booléen ; (les opérandes peuvent se réduire à un simple mode, mais ils peuvent aussi être constitués de plusieurs modes dans le cas de propositions fermées).

### 6.2. Forceurs

Le mode du résultat d'une confrontation qui est un forceur est le mode spécifié par le déclareur qui constitue son opérande gauche en notation infixée Le traitement du *:* d'un forceur se réduit donc à la suppression de l'opérande q' se trouve au sommet de la pile de travail.

Il se pose un léger problème à propos des forceurs neutres, car certains d'entre eux ne peuvent être différenciés de paires de bornes qu'après différenciation des appels et des tranches (voir les remarques après le paragraphe 6.6) Cependant un tel forceur ne peut être l'opérande d'une formule ; ce problème n'influe donc pas sur le seul processus d'identification.

6.3. Affectations

Le symbole d'affectation := joue le rôle d'un opérateur dyadique ; le mode du résultat se déduit de celui de l'opérande gauche (destination de l'affectation) par un nombre, éventuellement nul, de déprocédurages et doit obligatoirement commencer par "repère de" (le contexte est dit *mou*). Cet opérande gauche est un tertiaire ; il peut donc y avoir équilibrage, mais il se réduit à une suite d'alternatives (la destination d'une affectation ne peut être une proposition collatérale). Pour le processus d'identification seule importe la connaissance du mode du résultat. Soit $m_1$, $m_2$, ..., $m_n$ les entiers représentant les modes des propositions unitaires donnant lieu à équilibrage (éventuellement à plusieurs niveaux d'imbrication). Un algorithme permettant de trouver le mode de la destination peut se décrire de la manière suivante :

nous dirons qu'un mode est *hissable* s'il caractérise l'une des bases qui peuvent être hissées ; nous appellerons *déprocédure* d'un mode le mode obtenu en le déprocédurant au maximum (le déprocéduré de chacun des modes $m_i$, $1 \leq i \leq n$, qui n'est pas hissable doit être le mode d'un nom, sinon c'est une erreur) ; enfin nous appellerons *longueur* d'un mode le nombre maximum de dérepérages ou déprocédurages qu'il peut subir ;

ces définitions étant posées, soit $m$ un mode courant initialisé avec $m_1$ ; $m$ est confronté avec $m_i$, $2 \leq i \leq n$, comme suit :

- si $m$ est hissable, $m$ prend la valeur de $m_i$, sinon
- si $m_i$ est hissable, $m$ reste inchangé, sinon
- si la longueur du déprocéduré de $m$ est inférieure à la longueur du déprocéduré de $m_i$ alors $m$ reste inchangé, sinon il prend la valeur de $m_i$.

A la fin, $m$ représente le mode cherché, c'est-à-dire le mode de la destination d'une affectation ; si ce mode est hissable il y a erreur.

6.4. Sélections

Le symbole *de* d'une sélection se comporte comme un opérateur dyadique dont l'opérande gauche est un sélecteur de champ et non un mode ; l'opérande droit est un secondaire dont le contexte est faible, (il peut donc y avoir équilibrage mais non proposition collatérale), dont le mode doit commencer par "structure avec" ou "repère de structure avec". Ce mode peut être obtenu par

dérepérage et déprocédurage ; dans le cas du dérepérage le "repère de" qui précède "structure avec" dans le mode doit être conservé.

L'algorithme décrit pour les modifications en contexte ferme peut donc être utilisé en y supprimant le procédurage et ce qui concerne les unions, et en y remplaçant dans le test d'égalité la phrase : "si le mode a priori et le mode a posteriori sont égaux" par : "si le mode a priori est celui d'une valeur structurée ou celui d'un nom de valeur structurée".

Dans le cas de l'équilibrage, il suffit de considérer, dans la suite de modes possibles, celui de la première modande qui ne soit pas hissable et de lui appliquer l'algorithme précédent.

Le mode du résultat est le mode du champ sélectionné par le sélecteur constituant l'opérande gauche de *de* dans le cas où il s'agit d'une valeur structurée, et ce mode précédé de "repère de" dans le cas contraire (nom d'une valeur structurée). La représentation du système d'équations est conçue pour faciliter cette recherche (voir chapitre 3, paragraphe 2.2).

### 6.5. Appels

Un appel (ou une tranche) est caractérisé par la succession d'une parenthèse généralisée fermante ( ), *fin*, *fsi*, *sac*) ou d'un identificateur et d'une parenthèse généralisée ouvrante ( (, *début*, *si*, *cas*). Ceci n'est pas rigoureusement exact pour les répétitions dynamiques dans les notations de format (il est cependant possible de considérer qu'une telle répétition est un appel de procédure d'identificateur *n* ; en outre, les formats peuvent faire l'objet d'un traitement séparé et être supprimés de la chaîne source).

L'opérateur d'appel de procédure peut être considéré comme un opérateur à deux opérandes. Seul est intéressant pour l'identification l'opérande gauche qui est un primaire en contexte ferme. Il peut y avoir équilibrage ; le mode a posteriori de ce primaire devant commencer par "procédure avec", le traitement de l'équilibrage est simplifié : il suffit, comme dans le paragraphe précédent de considérer la première possibilité ne donnant pas lieu à hissage. D'autre part, dans l'algorithme du paragraphe 4, la phrase déjà citée est remplacée par : "si le mode a priori est celui d'une procédure avec paramètres".

Le mode du résultat se déduit directement du mode a posteriori du primaire, en prenant en considération la dernière inconnue de l'équation représentant ce mode. Les deux opérandes sont dépilés, ce qui, pour les paramètres effectifs est

un peu plus complexe. (Nous verrons au paragraphe 7.3.5 qu'ils peuvent être traités comme les champs d'une structure).

6.6 Tranches

Cette fois, parmi les parenthèses généralisées fermantes, nous devons considérer le dernier symbole guillemet d'une notation de chaîne, car le mode d'une telle notation est celui d'une valeur multiple. (Comme pour les appels, ceci est inexact dans une notation de format et les notations de chaîne ont pu être éliminées dans une phase d'édition).

L'opérateur de tranche peut être considéré comme un opérateur dyadique. Pour l'identification non seulement l'opérande gauche doit être pris en compte mais aussi l'opérande droit, c'est-à-dire l'indexeur. La détermination du mode a posteriori du primaire se fait de façon analogue à celle des sélections en remplaçant, dans l'algorithme du paragraphe 4, la phrase déjà citée par : "si le mode a priori est celui d'une valeur multiple ou celui d'un nom de valeur multiple".

Le mode du résultat se déduit du mode a posteriori du primaire et du nombre d'indices de l'indexeur ; en supposant que l'indexeur se trouve dans la pile de travail avec ses parenthèses (c'est-à-dire ses crochets) et ses virgules, un indice sera représenté par un mode (ou une conditionnelle ou une sérielle) figurant entre "[" et "," ou bien entre "," et "," ou bien entre "," et ")". De même que dans le cas d'une sélection, nous introduirons un "repère de" le cas échéant.

Remarques :

Après avoir identifié une tranche, les parenthèses entourant l'indexeur peuvent être remplacées par des crochets, si nécessaire.

Ce n'est qu'après avoir différencié les tranches des appels qu'il est possible de reconnaître certains forceurs neutres en paquet sans parenthèses (extension R.9.2. d).

Exemple : les deux déclarations d'identités

*proc (proc, proc, ...) f = fant* et *[,,] réel t = fant*

permettent d'avoir la tranche *t ( : p + q, : z * s, ...*

et l'appel *f ( : a + b, : c / d, ...*

### 6.7. Neutralisation

Nous rappelons qu'un point-virgule qui ne sépare pas deux paramètres doit provoquer la neutralisation de ce que nous pouvons considérer comme son opérande gauche. Pour l'identification, la neutralisation se réduit au dépilage de l'opérande qui se trouve au sommet de la pile de travail.

## 7 - CONTROLE DE TOUS LES MODES UTILISES DANS UN PROGRAMME ET TRAITEMENT DES MODIFICATIONS DANS LEUR ENSEMBLE

En même temps que l'identification des opérateurs il est possible de procéder à une vérification complète de la correction de tous les modes utilisés dans un programme et d'insérer, dans la chaîne postfixée, les opérateurs implicites correspondant aux modifications de modes détectées.

### 7.1. Formules simples

Nous appellerons formules simples les formules dont les opérandes ne donnent pas lieu à équilibrage. Après avoir identifié l'opérateur, il n'y a pas, dans ce cas, de vérification de mode à faire puisqu'il existe une suite de modifications élémentaires autorisées en contexte ferme qui permette d'assurer le passage du mode a priori au mode a posteriori ; nous pouvons donc insérer dans la chaîne postfixée les opérateurs qui correspondent aux modifications qui se trouvent dans le tableau *modif*. Pour réaliser cette insertion, nous devons déterminer l'opérande auquel s'applique la suite de modifications en tenant compte des propriétés de la notation postfixée.

Nous considérerons que tous les éléments, opérateurs et opérandes, de la chaîne postfixée représentant une formule simple sont séparés entre eux par des "trous" ; pour un opérateur en cours de traitement, la suite de modifications sera toujours à insérer dans le premier "trou" libre à gauche de cet opérateur :

- dans le cas d'un opérateur monadique ce trou sera toujours situé immédiatement à gauche de l'opérateur,
- dans le cas d'un opérateur dyadique, le trou correspondant aux modifications à effectuer sur son opérande droit (en notation infixée) se trouvera immédiatement à gauche de cet opérateur dans la chaîne postfixée, celui correspondant aux modifications à effectuer sur son opérande gauche (toujours en notation infixée) sera le premier trou libre situé sur la gauche de cet opérateu

dans la chaîne.

Exemple :

Soit l'expression *b / (c+d) * (-d-e)*

qui s'écrit en postfixée (en supposant que les priorités sont standards)

*b c d + / d - e - **

et qui donne lieu aux suites de modifications *m1* (resp. *m2*) pour l'opérande gauche (resp. droit) de l'opérateur *+*, *m3* (resp. *m4*) pour l'opérande gauche (resp. droit) de l'opérateur */*, *m5* pour l'opérande de l'opérateur monadique *-*, *m6* (resp. *m7*) pour l'opérande gauche (resp. droit) de l'opérateur dyadique *-* et *m8* (resp. *m9*) pour l'opérande gauche (resp. droit) de l'opérateur * ; après insertion de ces suites de modifications la chaîne postfixée est la suivante :

*b m3 c m1 d m2 + m4 / m8 d m5 - m6 e m7 - m9 **

Dans le cas de la modification *procédurer*, une partie de la chaîne postfixée doit être transformée en sous-programme ; il est donc nécessaire de délimiter cette partie de chaîne par deux marqueurs (ce qui correspond au fait que cette portion de programme ne sera pas exécutée en séquence ; voir aussi R.8.2.3.2). Cette délimitation peut s'effectuer en plaçant le marqueur dans le premier trou libre sur la gauche après avoir inséré la modification procédurer à sa place ; cette dernière joue le rôle du second marqueur. Quand il y a plusieurs procédurages à la suite le trou où est placé le premier marqueur doit toujours être considéré comme libre ; il doit aussi être considéré comme tel pour placer les opérateurs implicites de modification.

Pour l'exemple précédent, si la suite de modifications *m4* contient un procédurage, la chaîne postfixée obtenue serait

*b m3 ≠≠ c m1 d m2 + m4 / ...*

Enfin pour procédurer l'ensemble de la formule, nous considérerons qu'il y a un trou de procédurage en tête de la chaîne postfixée correspondant à la formule.

Nous utiliserons une structure de liste pour mettre en oeuvre une telle méthode ; les trous libres peuvent d'ailleurs être chainés entre eux au fur et à mesure que la chaîne est traitée. Quand la suite de modifications est vide ou épuisée, les trous doivent être bouchés.

7.2. Formules donnant lieu à équilibrage

Nous ne considérerons que le cas d'un opérateur monadique, celui des opérateurs dyadiques s'en déduisant facilement. L'identification de l'opérateur mis en jeu permet de déterminer le mode contextuel de la proposition fermée constituant son opérande ; ensuite il y a lieu d'insérer dans la chaîne post-fixée les modifications nécessaires tout en contrôlant la correction des modes des propositions unitaires constituantes. Nous supposerons que les séparateurs de propositions unitaires (virgule et barre verticale) et la parenthèse fermante jouent le rôle d'opérateurs, c'est-à-dire qu'ils sont précédés d'un trou ; d'autre part une parenthèse fermante ne sera jamais suivie d'un trou car une proposition fermée n'est pas une modande ; enfin rappelons qu'il y a un trou de procédurage en tête de chaque modande.

L'identification ayant été faite, nous pouvons supposer que toutes les propositions unitaires sont en contexte fort. En utilisant le symbolisme introduit au paragraphe 3 et en supposant que la proposition fermée se trouve dans la pile de travail comme pour l'identification, l'algorithme proposé pour contrôler les modes est le suivant :

une pile auxiliaire *p2* est utilisée pour décomposer le mode a posteriori ;

un compteur de parenthèse est initialisé à zéro, puis les éléments de la proposition fermée sont examinés l'un après l'autre à partir du sommet de la pile ;

- si l'élément examiné est une parenthèse fermante le compteur de parenthèses est incrémenté de un et le mode qui se trouve au sommet de *p2* est décomposé comme suit :

  . si ce mode est celui d'une valeur multiple il est remplacé par celui des éléments de cette valeur,

  . si ce mode est celui d'une valeur structurée il est remplacé par les modes des champs de cette valeur ;

ensuite l'élément suivant est examiné ;

- si l'élément examiné est une virgule et si le mode qui se trouve au sommet de la pile *p2* résulte de la décomposition d'une valeur structurée, ce dernier mode est dépilé et l'élément suivant est examiné ;

- si l'élément examiné est une parenthèse ou une accolade ouvrante, le compteur de parenthèse est décrémenté de un ; si ce compteur atteint la valeur zéro, l'algorithme est terminé, sinon l'élément suivant est examiné ;

- si l'élément examiné est une accolade fermante, le compteur est incrémenté de un, puis l'élément suivant est examiné ;

- la barre verticale est ignorée ;

- dans tous les autres cas l'élément examiné représente un mode ; si ce mode est fortement modifié à partir du mode qui se trouve au sommet de la pile auxiliaire *p2* alors les modifications élémentaires sont insérées et l'élément suivant est examiné, sinon il y a erreur (les modes utilisés ne sont pas compatibles).

## 7.3. Traitement des opérateurs généralisés

Les opérateurs généralisés nécessitent aussi l'insertion d'opérateurs implicites de modification et un contrôle sur les modes.

### 7.3.1. Relations de conformité et d'identité

Pour une relation de conformité le mode de l'opérande droit (en notation infixée) peut être quelconque et ne donne lieu à aucune modification.

L'opérande gauche est un tertiaire en contexte mou (le déprocédurage seul est autorisé) dont le mode a posteriori doit commencer par "repère de". Il peut donc y avoir équilibrage. Dans ce cas la détermination du mode de ce tertiaire est réalisée de la même manière que dans le cas de la destination d'une affectation (voir paragraphe 6.3) ; ensuite cet opérande est traité de façon normale par la méthode du paragraphe 7.2.

Dans le cas d'une relation d'identité, les modes a posteriori des deux opérandes doivent être les mêmes et commencer par "repère de". Il y a en plus d'un équilibrage possible pour chacun d'eux, un équilibrage pour les deux opérandes ; ainsi l'un peut être en contexte mou et l'autre en contexte fort ou l'inverse. La détermination du mode a posteriori des deux opérandes peut être réalisée à l'aide de la méthode du paragraphe 6.3 appliquée à l'ensemble des alternatives de chacun de ces opérandes. Si ce mode est hissable, il y a erreur sur les modes utilisés. Ensuite les deux opérandes sont traités séparément pour le contrôle de mode.

### 7.3.2. Forceurs

L'opérande gauche étant un déclareur, il n'y a aucune modification à mettre en jeu, mais il convient de faire disparaitre le trou correspondant. L'opérande droit se trouve en contexte fort et il peut être traité de la même façon que les

opérandes des formules ; en particulier, en cas d'équilibrage, la méthode du paragraphe 7.2 s'applique directement.

7.3.3. Affectations

Le mode a posteriori de la destination est déterminé selon la méthode du paragraphe 6.3, puis les deux opérandes sont traités comme ceux des autres formules.

7.3.4. Sélections

Le sélecteur n'est pas une modande, il ne donne donc lieu à aucune modification et le trou correspondant doit être supprimé. La détermination du mode a posteriori de l'opérande droit étant réalisée (paragraphe 6.4), l'insertion des opérateurs de modification et le contrôle des modes ne posent pas de problèmes particuliers.

7.3.5. Appels

Le mode a posteriori de l'opérande gauche (primaire) est déterminé (paragraphe 6.5) puis traité comme l'opérande gauche d'une formule. Nous considèrerons que l'opérande droit (paramètres effectifs en paquet) peut être assimilé à une proposition collatérale de mode structurée ; les paramètres effectifs seront donc traités par la méthode du paragraphe 7.2 en plaçant dans la pile auxiliaire *p2* les modes a posteriori des paramètres ; ces modes se déduisent directement du mode a posteriori du primaire de l'appel.

7.3.6. Tranches

L'opérande gauche sera traité de la même manière que le primaire d'un appel. L'opérande droit (indexeur) peut être assimilé à une proposition collatérale de mode a posteriori "rang d'entier" ; les séparateurs des propositions unitaires de cette dernière sont un peu particuliers ( ,, ;, *apd* ) et ces propositions unitaires peuvent être vides. Il est aussi possible de considérer, pour le contrôle des modes, : et *apd* comme des opérateurs dyadiques à opérandes entiers et à résultat entier.

Nous remarquerons que les rangeurs des déclareurs de valeurs multiples sont assimilables à des indexeurs.

### 7.3.7. Neutralisation

Le point virgule peut être considéré, dans certains cas, comme un opérateur de neutralisation, c'est-à-dire un opérateur monadique dont l'opérande gauche a pour genre a posteriori neutre. Il en est de même du deux points d'un forceur neutre. Il y a deux cas à distinguer :

- la neutralisation des confrontations qui n'entraine aucun contrôle de mode et pour laquelle l'opérateur implicite à insérer correspond au dépilage de la valeur de la modande neutralisée ;

- la neutralisation des autres modandes qui nécessite un contrôle de mode et l'insertion éventuelle de modifications élémentaires autres que la neutralisation ; en pratique la modande est déprocédurée et dérepérée au maximum ; si le genre obtenu est le "neutre" il n'y a pas lieu de neutraliser ; si ce genre est tel qu'en lui retirant les "rang de" et les "repère de" qui le commence on obtient le mode d'une procédure sans paramètres il y a erreur (voir règle R.8.2.1. b deuxième alternative) ; dans les autres cas les modifications à insérer, sont, outre la neutralisation, les modifications qui ont permis de déprocédurer au maximum (ces modifications sont des dérepérages ou des déprocédurages et la dernière doit être un déprocédurage).

### 7.4. Traitement des propositions conditionnelles et instructions d'itération

Une forme postfixée pour les propositions conditionnelles (contractées ou non) est définie ; cette forme traduit le fait qu'il y a équilibrage et que *si*, *alsi* et *sinsi* soient, en ce qui concerne les modes et modifications, le rôle d'opérateur monadique dont l'opérande se trouve en contexte fort et est de mode a posteriori booléen. De façon analogue le symbole *cas* pour les *propositions cas* joue le rôle d'opérateur monadique à opérande en contexte fort et de mode entier. (Pour les propositions cas de conformité il n'y a pas de modifications possibles).

Exemples : *si b alors p sinon q fsi* peut devenir *b si { p | q }*
*cas i dans u, v, w sinon z fsac* peut devenir *i cas { u | v | w }*
*si b alsi p alors q sinon r fsi* peut devenir *b si q alsi { q | r }*

Pour les instructions d'itération *depuis*, *pas*, *jusqu'à*, (resp. *tantque*) sont assimilables à des opérateurs monadiques à opérandes en contexte fort et de mode entier (resp. booléen). *pour i* est pris en considération dans le processus d'identification des identificateurs. Enfin *faire* doit neutraliser son opérande

droit (voir paragraphe 7.3.7).

Remarquons que par définition, le genre a priori d'une instruction d'itération est neutre et que ce n'est pas une modande.

## 7.5. Traitement des déclarations d'identité et d'opération

Quand il n'y a pas d'extension définie par la règle R.9.2.a ou la règle R.9.2.e, la correction du mode du paramètre effectif doit être vérifiée et les modifications requises sont insérées. Une telle déclaration d'identité d'opération peut être assimilée à un forceur dont le mode est celui du paramètre formel ou de l'affiche.

# Annexe au CHAPITRE I

## Liste des protonotions simples

A,ACTION,ACTUAL,ADIC,ALEPH,ALIGNMENT,AND,ASSIGNATION,

B,BALANCE,BASE,BASIC,BIT,BOOLEAN,BOUND,BOUNDSCRIPT,BOX,BUT,BY,

C,CALL,CAPTION,CAST,CHAIN,CHARACTER,CHOICE,CLAUSE,CLOSED,COERCEND,COHESION,
COLLATERAL,COLLECTION,COMMENT,COMPLETER,COMPLEX,CONDITION,CONDITIONAL,
CONFORMITY,CONFRONTATI,

D,DECLARATION,DECLARATOR,DECLARER,DENOTATION,DEPROCEDURED,DEREFERENCED,DESTI-
NATION,DIGIT,DYADIC,DYNAMIC,

E,EIGHT,ELSE,END,EXIT,EXPONENT,EXPRESSION,EXTRA,

F,FIELD,FIRM,FIVE,FLEXIBLE,FLIPFLOP,FLOATTING,FORMAL,FORMAT,FORMULA,FOUR,
FRACTIONAL,FRAME,

G,GENERATOR,GLOBAL,GO,GOMMA,

H,HIP,HIPPED,

I,IDENTIFIER,IDENTITY,IMAGE,INDEXER,INDICATION,INSERT,INSERTION,INTEGRAL,INTER-
LUDE,ITEM,

J,JUMP,

K,

L,LABEL,LEAVING,LETTER,LIBRARY,LIST,LITERAL,LOCAL,LONG,LOOSE,LOWER,LY,

M,MODE,MONADIC,MOULD,

N,NEW,NIHIL,NINE,NOT,NUMBER,NUMERAL,

O,OF,ON,ONE,OPEN,OPERAND,OPERATION,OPERATOR,OPTION,OUT,

P,PACK,PACKAGE,PARALLEL,PARAMETER,PART,PARTICULAR,PATTERN,PHRASE,PICTURE,
PLAIN,PLAN,PLUPLUMINU,POINT,POSTLUDE,POWER,PRELUDE,PRIMARY,PRIORITY,
PROCEDURE,PROCEDURED,PROGRAM,PROPER,

Q,QUOTE,

R,RADIX,RANGE,REAL,REFERENCE,RELATION,RELATOR,REPLICATABLE,REPLICATED,REPLICA-
TION,REPLICATOR,ROUTINE,ROW,ROWED,ROWER,

S,SECONDARY,SELECTION,SELECTOR,SEPARATED,SEPARATOR,SEQUENCE,SEQUENCED,SEQUENCING,
SERIAL,SEVEN,SIGN,SINGLE,SIX,SKIP,SLICE,SOFT,SOME,SOURCE,SPECIAL,STAG-
NANT,STANDARD,STATEMENT,STRICT,STRING,STRONG,STRUCTURE,STRUCTURED,SUB-
SCRIPT,SUITE,SUPPRESSIBLE,SYMBOL,SYNTACTIC,

T,TEN,TERTIARY,THE,THEN,THREE,TIME,TO,TOKEN,TRAIN,TRANSFORMAT,TRIMMER,TRIM-
SCRIPT,TWO,

U,UNION,UNIT,UNITARY,UNITED,UPPER,

V,VACUUM,VARIABLE,VIRTUAL,VOID,VOIDED,

W,WEAK,WIDENED,WITH,

X,

Y,

Z,ZERO,

# ANNEXE AU CHAPITRE 3

```
co description d'un algorithme de construction d'un système d'équations permettant de représenter les modes
   spécifiés par des déclareurs ; voir chapitre 3, paragraphe 2.3 co
début ent imax, jmax, kmax ; lire ((imax, jmax, kmax, à la ligne)) ;
   [1:jmax] ent système, [1:kmax] ent accès, [1:lmax] struct (ent code, mode) indicateurs,
   ent jmin = fant, kmin = fant, lmin = fant co ces valeurs doivent être fixées en tenant compte de la partie
                                                du système représentant les modes primitifs et standards co ;
   ent i:=0, j:=jmin, k:=kmin, fond de p:=0, i1, j1,
   ent rep = -11, rang = -12, proc = -13, procavec = -14, struct = -15, union = -16, repstrung = -17,
      par = code ("("), rap = code (")"), égale = code ("="),
   chaîne us, bool déjalue =faux, tableau:=faux,
   ent xneutre = fant, xent = fant, xréel = fant co, xlent = fant, xlréel = fant co,
      co ces entiers représentent les modes primitifs correspondants et le fant doit être remplacé par une
         valeur adéquate co ;
   proc empiler co dans la pile p co = (ent m) : p [i plus 1] := m,
   proc erreur = (chaîne ch) : (imprimer ("erreur.:." + ch, à la ligne) ; exit),
   proc code = (chaîne cha) ent : co cette procédure fournit la valeur entière codant son paramètre ; il est
                                          sans intérêt de l'expliciter co fant,
   proc identificateur = (chaîne cha) bool : co cette procédure fournit la valeur vrai si son paramètre est
                                                un identificateur co fant,
   proc indicateur = (chaîne cha) bool : co même chose pour un indicateur de mode co fant,
   proc recherche = (pour k1 depuis kmin jusqua k-1 faire
                        si p [i] égale k co voir annexe au chapitre 4 co
                           alors (p [i] := k1, j:=j1, k moins 1) ; arrêt
                        fsi ;
                     arrêt : fant) ;
```

```
proc bool neutre = co cette procédure sert à détecter l'absence de résultat pour le déclareur de procédure co
                   us ≠ "rep" et us ≠ "struct" et us ≠ "proc" et us ≠ "union"
                   et us ≠ "[" et non indicateur (us) ;
proc pos i1 = co cette procédure fait prendre à i1 l'indice de l'élément qui précède la première parenthèse
                ouvrante dans p co :
                (i1:= i  ; tantque p [i1moins 1] ≠ par faire fant ;
                 i1 moins 1) ;
proc prouction = co cette procédure réalise l'écriture dans le système d'un second membre commençant par
                   procavec, struct ou union co :
                  (j1 := j ; ent pi1 = p[i1] ; système [j plus 1] := pi1 ;
                   si pi1 = struct
                      alors système [j plus 1] := (i - i1) ÷ 2 - 1 ;
                            pour l depuis i1 + 3 pas 2 jusquà i - 1 faire
                                 système [j plus 1] := p[l] ;
                            pour l depuis i1 + 2 pas 2 jusquà i-2 faire
                                 système [j plus 1] := p[l] ;
                            jusquà système [j1 + 2] faire
                                 système [j plus 1] := 0 ; co réservation de place pour les repères de
                                                             champs co
                   sinon système [j plus 1] := i - i1 - 2 ;
                            pour l depuis i1 + 2 jusquà i faire
                                 (ent pl = p [l] ; si pl ≠ rap alors système [j plus 1] := pl fsi)
                   fsi ;
```

```
                accès [(p[(i := i1)] := k plus 1] := j1 + 1
            ),
proc repère = (ent pi) co cette procédure traite le cas d'un repère de valeur structurée ou multiple co :
            début ent api = accès [pi] ; ent sapi = système [api] ;
                si sapi ≥ 0
                    alors co rep pi et pi est non définie ; il y a donc lieu de signaler qu'il y
                            eu un rep sur un indicateur non encore défini co
                        ent man1 := pi, man2 ;
                        e : si (man2 := accès [man1]) = 0
                            alors accès [man1] := max ent
                            sinon man1 := man2 ; e
                            fsi
                sinsi sapi = struct
                    alors co repère de valeur structurée co
                        ent n = système [api + 1] ;
                        si système [api + 2 * (n+1)] = 0
                            alors pour ja depuis api + 2 jusquà api + n + 2 faire
                                ( empiler (ja + n) ; empiler (repstrung) ; empiler (système [ja]))
                        fsi
                sinsi sapi = rang
                    alors co repère de valeur multiple co
                        si système [api + 2] = 0
                            alors empiler (api + 2) ; empiler (repstrung) ; empiler (système [api + 1])
                        fsi
                fsi ;
```

```
co commencer ici co
suivant : si non déjalue alors lire (us) fsi ; déjalue := faux ;
          si us = "rep" ou us = "struct" ou us = "union" ou us = "(" ou us = "=" ou identificateur (us)
            alors empiler (code (us)) ;
                  suivant
         fsi ;
         si us = "proc"
           alors (lire (us), déjalue := vrai) ;
                 si us = "(" alors empiler (procavec) ; suivant fsi ;
                 empiler (proc) ;
                 si neutre alors empiler (xneutre) ; réduction
                           sinon suivant
                 fsi
        fsi ;
        si us = "[" alors tableau := vrai ; empiler (rang) ; suivant fsi ;
        si us = "," alsi tableau alors empiler (rang) fsi ; suivant fsi ;
        si us = "]" alsi tableau alors tableau := faux ; suivant sinon erreur ("tableau") fsi ;
        si indicateur (us)
          alors ent codus = code (us) ;
                pour l1 jusqà l faire
                si code de indicateurs [l] = codus
                  alors empiler (mode de indicateurs [l] ) ;
                        réduction
```

```
                    sinon indicateurs [l plus 1] := (codus, k plus 1) ;
                          (empiler (k), accès [k] := 0) ;
                          réduction
                    fsi
          fsi ;
co traitement des parenthèses fermantes co
          si us = ")"
             alors empiler (rap) ; pos i1 ;
                   si p [i1] = procavec
                      alors (lire (us), déjalue := vrai) ;
                            si neutre
                               alors empiler (xneutre)
                               sinon suivant
                            fsi
                   fsi ;
                   prouction ; test mini
          fsi ;
co dans tous les autres cas co suivant ;

test mini : recherche co dans le système d'une équation définissant le même mode que l'entier qui se
                    trouve au sommet de la pile et remplacement éventuel de ce dernier par cette
                    équation co
réduction : si i = fond de p + 1
               alors co cas a) ; il faut alors associer le mode représenté par l'entier qui se trouve
                       au sommet de p à un déclareur "isolé" co
```

```
        i := fond de p ; co réinitialisation pour un autre déclareur co
        suivant
sinsi i = fond de p + 3 et p [i-1] = égale
    alors co cas b) ; il s'agit d'une déclaration de mode co
        ent xa = p [i-2], xb = p [i] ;
        si système [accès [xa]] < 0
           alors erreur ("""double.déclaration""") co i co
        sinsi système [accès [xb]] ≥ 0
           alors accès [xb] := xa                    co ii co
           sinon ent man1 := xa, man2 ;              co iii co
                 e : si man2 := accès [man1] ;
                        accès [man1] := xb ;
                        ent mm = (man1 := man2) ;
                        mm = 0 ou mm = maxent
                        alors arrêt
                        sinon e
                     fsi ;
            arrêt : si man1 = maxent
                       alors repère (xb)
                    fsi
        fsi ;
        i := fond de p ;
    sinon co cas c) co
```

```
        ent pi = p[i], pi1 = p[i-1] ;
        proc équa = (ent préfixe) co cette procédure écrit une équation dont le second membre
                                     commence par "préfixe" (rep, proc ou rang) co :
                    début j1 := j ;
                          système [(accès[(p[i moins 1] := k plus 1)] := j plus 1)] := préfixe ;
                          système [j plus 1] := pi
                    fin ;
        si pi1 = rap
           alors prouction co cas d'une procédure avec paramètres co ; test mini
        sinsi pi1 = proc
           alors équa (proc) ; test mini
        sinsi pi1 = rang
           alors équa (rang) ; système [j plus 1] := 0 ; test mini
        sinsi pi1 = rep
           alors équa (rep) ; recherche ; repère (p[i])
        sinsi pi1 = repstrung
           alors équa (rep) ; recherche ; ent pi = p[i] ; système [p[i-1]] := pi ;
                 i moins 2 ; repère (pi)
        fsi
    fsi ;
    suivant
fin
```

# ANNEXE AU CHAPITRE 4

```
co vérification de la condition sur les déclarations de mode ; voir chapitre 4, paragraphe 1.2 co
co dans ce qui suit les identificateurs non déclarés ont la même signification que dans l'annexe
   précédente co

début mode doublet = struct (ent mode, marque) ;
      [1 : imax] doublet pile,
      pour l1 depuis lmin jusquà l faire
         début pour k1 depuis kmin jusquà k faire
                  (co démarquage des inconnues éventuellement marquées co
                   rep ent ak1 = accès [k1] ; si ak1 < 0 alors ak1 := -ak1 fsi) ;
               ent modind = mode de indicateurs [l1];
               si système [accès [modind]] ≥ 0
                  alors erreur ("indicateur.non.défini")
                  sinon i := 0 ;
                        empiler ((modind, 0)) ;
               continuer : si i = 0
                              alors co la condition de mode est vérifiée pour l'indicateur en cours de
                                       traitement, il faut donc traiter le co suivant
                              sinon ent mode courant = mode de pile [i],
                                       marque courante = marque de pile [i] ;
                                    rep ent amc = accès [mode courant] ;
                                    ent préfixe = système [amc] , longueur = système [amc + 1] ;
                                    proc vérif = (ent marque) :
```

```
                pour k depuis amc + si préfixe = procavec ou préfixe = union
                                       alors 2
                                    sinsi préfixe = struct
                                       alors longueur + 2
                                       sinon 1
                                    fsi
                          jusquà amc + si préfixe = procavec ou préfixe = union
                                       alors longueur + 2
                                    sinsi préfixe = struct
                                       alors 2 * longueur + 2
                                       sinon 1
                                    fsi
                          faire (ent sk = système [k] ;
                                 si sk = modind
                                    alors erreur ("condition.de.mode")
                                    sinon empiler ((sk, marque))
                                 fsi) ; co fin de vérif co
i moins 1
si amc < 0
   alors co l'inconnue correspondante est marquée, il faut donc co
                continuer
   sinon amc := - amc co marquage co
```

```
                    si mode courant ≤ kmin co primitif ou standard co
                       ou préfixe = procavec
                       alors fant
                    sinsi préfixe = struct
                       alors si marque courante ≠ rep
                                alors vérif (struct) fsi
                    sinsi préfixe = rep
                    alsi marque courante ≠ struct
                       alors vérif (rep)
                       sinon vérif (marque courante)
                    fsi ;
                    continuer
                fsi
            fsi
suivant  : fant
        fsi
fin
```

```
co développement des unions ; voir chapitre 4, paragraphe 1.3 co
co de même, dans ce qui suit les identificateurs non déclarés ont la même signification que précédemment co
début proc transférer = (ent syst) co transférer les inconnues du second membre commençant à "syst" dans la
                                    pile co
                        pour j1 depuis syst +2 jusquà syst + système [syst+1] +2 faire
                            p [i plus 1] := système [j1] ;
    pour k1 depuis kmin jusquà k faire
        début rep ent ak1 = accès [k1] ;
            si système [ak1] = union
              alors bool b := faux ; i := 0 ; ent jaux := 0
                    transférer (ak1) ;
                    tantque i ≠ 0 faire
                        (ent api = accès [p[i]] ;
                         si système [api] = union
                            alors (transférer (api), b := vrai)
                            sinon système [j + 2 + (jaux plus 1)] := p[i]
                         fsi ;
                         i moins 1) ;
                    si b alors (système [(ak := j + 1)] := union, système [j + 2] := jaux) ;
                               j := j + jaux + 1 fsi
            fsi
        fin
fin
```

```
co réduction du système d'équations ; voir chapitre 4, paragraphe 1.4 co
        pour k1 depuis kmin jusquà k-1 faire
        pour k2 depuis k1 + 1 jusquà k faire
           si accès [k1] égale accès [k2]
              alors accès [k1] := accès [k2]
           fsi
co contrôle du non apparentement des composantes d'union, chapitre 4, paragraphe 1.5  co
début [1 : t co t a la valeur maximum prise par jaux lors du développement des unions co ] ent tab ;
      proc contrôler = (ent m1, m2) :
                          début ent tab2 := m2, a2, ent a1 = accès [m1] ;
                        test : si a1 = (a2 := accès [tab2])
                                  alors erreur ("apparentement")
                                  sinon ent sa2 = système [a2] ;
                                        si sa2 = rep ou sa2 = proc
                                          alors tab2 := système [a2 + 1]
                                        fsi
                                fsi
                          fin ;
      pour k1 depuis kmin jusquà k faire
          début ent ak1 = accès [k1] ;
                si système [ak1] = union
                alors co trier en éliminant les composantes égales co
                     ent aux := 0, ent ak1 n := ak1 + système [ak1 + 1] ;
```

```
                pour j1 depuis ak1 + 2 jusquà ak1n faire
                    début ent max := système [j1], jmax ;
                        éti : pour j2 depuis j1 + 1 jusquà ak1n + 1 faire
                                si max < système [j2]
                                    alors (max := système [j2], jmax := j2)
                                fsi ;
                              si max ≠ aux
                                alors système [jm] := système [j1] ;
                                      système [j1] := aux  := max
                                sinon (système [ak1 + 1] moins 1, système [jm] := 0) ;
                                      éti
                              fsi
                    fin ;
                ent l := i := 0 ;
                transférer (ak1) ;
phase 1  :      tantque i ≠ 0 faire
                    début rep ent pi = p[i] ; ent api = accès [pi] ;
                        si système [api] = proc
                          alors pi := système [api + 1]
                        sinsi système [api] = union
                          alors i moins 1
                                transférer (api)
                          sinon tab [l plus 1] := pi ;
```

```
                                        i moins 1
                                    fsi
                                fin ;
        phase 2  :  pour i1 jusquà l-1 faire
                        pour i2 depuis i1 + 1 jusquà l faire
                            (controler (tab [i1], tab [i2]), controler (tab [i2], tab [i1])
                    fsi
                fin
        fin
```

```
   co déclaration de l'opérateur égale qui permet de tester l'égalité des modes représentés par deux indices
      du vecteur d'accès ; voir chapitre 4, paragraphe 2.1 co
   priorité égale = 5 ;
   op égale = (ent mi, mj) bool :
      début mode doublet = struct (ent a, b) ;
            ent i := 0, j := 0 ; [1 : imax] doublet p1, [1 : jmax] doublet p2, ent  m = 0 ;
            proc empiler = (doublet doublet, rep [ ] doublet pile) :
                              pile  [ si pile :=: p1 alors i sinon j fsi plus 1]:= doublet ;
            proc dépiler = (rep [ ] doublet pile, co jusquà co doublet doublet) :
                              si pile :=: p1
                                 alors tantque p1 [i] ≠ doublet faire i moins 1
                                 sinon tantque p2 [j] ≠ doublet faire j moins 1
                              fsi ;
   co commencer ici co
   empiler ((mi, mj), p1) ;
iter : si i = 0
         alors vrai co les modes représentés par les opérandes de égale sont égaux co
         sinon ent ma = a de p1[i], mb = b de p1 [i] ;
               ent ama = accès [ma], amb = accès [mb] ;
               ent sma = système [ama], smb = système [amb] ;
               i moins 1 ;
               si ma = mb
                  alors iter
```

```
        sinsi pour k depuis j pas -1 jusquà 1 faire
                si ent mak = a de p2[k], mbk = b de p2[k] ;
                   ma = mak et mb = mbk ou ma = mbk et mb = mak
                   alors vrai
                fsi ;
             faux . co le doublet (ma, mb) n'a pas encore été rencontré co
             vrai : vrai co ce même doublet a déjà été examiné co
          alors iter
        sinsi sma = smb
alors   si sma = rep ou sma = proc ou sma = rang
          alors (empiler ((système [ama + 1], système [amb + 1]), p1),
                 empiler ((ma, mb), p2)) ; iter
        sinsi sma = procavec
          alors ent la = système [ama + 1], lb = système [amb + 1] ;
                si la = lb
                   alors (pour l depuis 2 jusquà la + 1 faire
                             empiler ((système [ama + l], système [amb + l]), p1)
                             empiler ((ma, mb), p2)) ; iter
                fsi
        sinsi sma = struct
          alors ent la = système [ama + 1], lb = système [amb + 1] ;
                si la = lb
                alsi pour l depuis 2 jusquà la + 1 faire
```

```
            si système [ama + l] ≠ système [amb + l]
               alors faux
            fsi ;
          vrai . co les sélecteurs de même rang sont identiques co
          faux : faux co il existe au moins deux sélecteurs de même rang différents co
        alors (pour l depuis la + 2 jusquà 2 * la + 1 faire
                 empiler ((système [ama + l], système [amb + l]), p1)
                empiler ((ma, mb), p2)) ; iter
    fsi
sinon co sma = union co
    (empiler ((m, m), p1), empiler ((m, m), p2)) ;
    ent la = système [ama + 1], lb = système [amb + 1] ;
    co puis empiler 2 * la * lb blocs co
    pour k depuis ama + 2 jusquà ama + la + 1 faire
      (pour l depuis amb + 2 jusquà amb + lb + 1 faire
         empiler ((système [k], système [l]), p1) ;
        empiler ((m, i) co séparateur de blocs co, p1)) ;
    pour k depuis amb + 2 jusquà amb + lb + 1 faire
      (pour l depuis ama + 2 jusquà ama + la + 1 faire
         empiler ((système [k], système [l], p1) ;
        empiler ((m, i), p1) ;
    (empiler ((ma, mb), p2), empiler (p1[i-1], p1)) ;
    empiler ((m, -1), p2) ; iter
fsi
```

```
        sinsi ma = m
           alors tantque a de p1 [i] ≠ m faire i moins 1 co dépiler un bloc co ;
                 co lancer le traitement du bloc suivant, c'est-à-dire co
                 (empiler (p1 [i - 1], p1), empiler ((m, -1), p2)) ;
                 iter
        fsi ;
        co ici se trouve le traitement d'équations non analogues co
test  : si i = 0
           alors faux co les modes représentés par mi et mj sont différents co
        sinsi a de p1[i] ≠ m
           alors co recherche du premier séparateur de bloc co
                 i moins 1 ; test co pour savoir si le fond de la pile p1 est atteint co
           sinon rep ent b = b de p1[i] ;
                 si a de p1 [b moins 1] ≠ m
                   alors co il existe encore au moins un doublet à examiner dans le bloc en cours co
                         dépiler (p2, (m, -1)) ;
                         empiler ((p1[b], p1) ; co essai du doublet suivant co
                         iter
                   sinon co il n'y a plus de doublet à examiner dans le bloc qui se trouve au sommet de p1 co
                         ((dépiler (p1, (m, m)) ; i moins 1), (dépiler (p2, (m, m)) ; j moins 1)) ;
                         test
                 fsi
        fsi
    fsi
fin
```

# ANNEXE AU CHAPITRE 5

```
co composition des modifications en contexte ferme, voir chapitre 5, paragraphe 4 co
[1 : jmax] ent modif ; ent j1 := 0, j2 := jmax-1 ;
ent dérepérer = fant co une valeur appropriée co, déprocédurer = fant, procédurer = fant, unir1 = fant, unir2 = fant ;
op estfermtmodifiéapd = (ent co mode co a priori, co mode co a posteriori) bool :
   début proc empiler = (ent elt, rep [ ]ent pile) :
                        si p1 :=: pile
                           alors p1 [i plus 1] := elt
                           sinon si j2 = j1+1 alors erreur ("modif") fsi ;
                                 modif [si elt = procédurer ou elt = unir1 ou elt = mettre en rang 1
                                           alors j2 moins 1
                                           sinon j1 plus 1
                                       fsi] := elt
                           fsi ;
      proc dereproc = (proc étiquette co sortie dans le cas où on ne peut ni dérepérer ni déprocédurer co) :
                        début ent aλ = accès [λ]
                              si système [aλ] = rep
                                 alors empiler (dérepérer, modif)
                              sinsi système [aλ] ≠ proc
                                 alors étiquette
                                 sinon empiler (déprocédurer, modif)
                              fsi ;
                              λ := système [aλ+1]
                        fin ;
```

```
        ent [1 : imax] ent p1, ent i := 0, ν ;
co commencer ici co
        ent λ := ν := a priori, μ := a posteriori
   test : si μ = λ co cas d'un système d'équations réduit co
            alors eureka
          sinsi système [accès [μ]] = union
            alors ent aλ = accès [λ], aμ = accès [μ] ;
                  si système [aλ] = union
                     alors si ent jaux ;
                             pour j depuis aμ+2 jusquà aμ+système [aμ+1] +2 faire
                               si système [aλ+2] = système [j]
                                  alors jaux := j ; mivrai
                               fsi ;
                               faux ;
                          mivrai : pour j depuis 3 jusquà système [aλ+1] +2 faire
                                     si système [aλ+j] ≠ système [aμ+jaux+j]
                                      alors faux
                                     fsi ;
                                      vrai . faux : faux
                                   alors empiler (unir2, modif) ; co règle R.3.2.4.1.d co
                                         eureka
                                   sinon décomposer co le mode a posteriori μ co
                                fsi
```

```
                    sinon dereproc (décomposer) ;
                          test

                fsi
          sinsi système [accès [μ]] = proc
              alors empiler (procédurer, modif) ;
                    μ := système [accès [μ]+1] ; co procédurage systématique co
              sinon dereproc (suivant)
          fsi ;
          test ;
décomposer : empiler (m, p1) ; co séparateur de blocs d'union co
            pour s depuis accès [μ] +2 jusquà accès [μ] +2 + système [accès [μ]+1] faire
               empiler (système [s], p1) ;
            empiler (unir1, modif) ;
  suivant : si i = 0
               alors tilt co il n'y a plus d'alternative pour unir co
            sinsi p1[i] = m
               alors i moins 1 ; si i = 0 alors tilt fsi ;
                     tantque modif [j2] ≠ unir1 faire j2 plus 1 ;
                     j2 plus 1
            fsi ;
            μ := p1 [i] ; i moins 1 ; co essai de l'alternative suivante co
            λ := ν ; j1 := 0 ; test ;
  eureka  : e1 : si j1 > 0 et j2 ≤ jmax
```

```
            alsi modif [j1] = déprocédurer et modif [j2] = procédurer
                 alors (j1 moins 1, j2 plus 1) ; e1   co suppression des procédurages et déprocédurages
                                                      successifs co
            fsi ;
        tantque j2 ≤ jmax faire co remise en ordre co
           ( modif [j1 plus 1] := modif [j2] ; j2 plus 1) ;
        vrai.
    tilt  : j1 := 0 ; faux
fin
```

```
co composition des modifications en contexte fort ; voir chapitre 5, paragraphe 5 co
ent élargir 1 = fant, élargir 2 = fant, élargir 3 = fant, élargir 4 = fant, co etc. co, hisser 1 = fant,
                                                                                    hisser 2 = fant,
    mettre en rang 1 = fant, mettre en rang2 = fant, mettre en rang 3 = fant, hisser 3 = fant, hisser 4 = fant,
                                                                                    hisser 5 = fant,
    vide = fant, fant = fant, saut = fant, nil = fant, xcompl = fant, xréel = fant co etc co ;
op estfortmodifiéapd = (ent a priori, a posteriori) bool :
   début proc empiler = co voir annexe précédente co fant ;
         proc dereproc = co voir annexe précédente co fant ;
         ent μ := a posteriori, λ := a priori ;
co commencer ici co
         si λ = saut
            alors empiler (si système [accès [μ]] = proc
                              alors hisser 1
                              sinon hisser 2
                           fsi, modif) ;
                  eureka
         sinsi λ = fant
            alors empiler (si système [accès [μ]] = union
                              alors hisser 3
                              sinon hisser 4
                           fsi, modif) ;
                  eureka
         sinsi λ = nil
```

```
            alors empiler (si système [accès  [μ]] = rep
                              alors hisser 5
                              sinon tilt
                           fsi, modif) ;
                  eureka
        sinsi λ = vide
              alors eti : si système [accès [μ]] = rang
                              alors empiler (mettre en rang 3, modif) ;
                                    eureka
                           sinsi système [accès [μ]] = proc
                              alors (μ := système [accès [μ]+1], empiler (procédurer, modif) ;
                                     eti
                              sinon tilt
                           fsi
        fsi ;
test  : si μ = λ
           alors eureka
        sinsi système [accès [μ]] = union
           alors si λ estfermtmodifiéapd  μ alors eureka sinon tilt fsi
        sinsi système [accès [μ]] = proc
           alors (empiler (procédurer, modif), μ := système [accès [μ]+1]) ;
                 test
        sinsi système [accès [μ]] = rang
           alors (empiler (mettre en rang 1, modif), μ := système [accès [μ]+1]) ;
                 test
```

```
sinsi système [accès [μ]] = rep
   alors ent μ1 = système [accès [μ]+1] ;
         si système [accès [μ1]] = rang
           alors ent μ2 := système [accès [μ1]+1], n := 1 ;
                 si système [accès [λ]] = rep
                   alors ent λ1 = système [accès [λ]+1] ;
                     eti : si λ1 = μ2
                                alors jusquà n faire empiler (mettre en rang 2, modif) ;
                                     eureka
                           sinsi système [accès [μ2]] = rang
                              alors ( μ2 := système [accès [μ2]+1], n plus 1)
                                   eti
                           fsi
                 fsi ;
                 dereproc (tilt) ; test
         fsi
sinsi μ = xcompl
   alors e : si λ = xréel
               alors empiler (élargir 2, modif) ; eureka
             sinsi λ = xent
               alors empiler (élargir1, modif) ;
                     empiler (élargir2, modif) ; eureka
               sinon dereproc (tilt) ; e
             fsi
sinsi μ = xréel
```

```
            alors e : si λ = xent
                        alors empiler (élargir1, modif) ; eureka
                        sinon dereproc (tilt) ; e
                      fsi
          sinsi μ = xbits
            alors e : si λ = xrangbool
                        alors empiler (élargir3, modif) ; eureka
                        sinon dereproc (tilt) ; e
                      fsi
          sinsi μ = xcat
            alors e : si λ = xrangcar
                        alors empiler (élargir4, modif) ; eureka
                        sinon dereproc (tilt) ; e
                      fsi
          co il conviendrait de traiter, pour ces quatre derniers cas les versions plus longues co
          fsi ;
eureka ;  co voir annexe précédente la fin de cette routine co fant
  fin
```

# BIBLIOGRAPHIE


[1] Van Wijngaarden, A. (Editeur), Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., *Report on the algorithmic language ALGOL 68, Mathematisch Centrum, MR101,* Amsterdam, Octobre 1969, et Numerische Mathematic, 14, 1969, pp. 79-218.

[2] Groupe Algol 68 de l'AFCET, Buffet, J., Arnal, P., Quéré, A. (Editeurs), *Définition du Langage Algorithmique ALGOL 68,*
*Présentation et Traduction française du Report on the algorithmic language ALGOL 68,*
à paraître chez Hermann, Paris.

[3] Pair, C.,
*Concerning the syntax of ALGOL 68,*
Algol Bulletin, AB 31.3.2, Mars 1970, pp.16-27.

[4] Bolliet, L.,
*Compiler writing techniques, dans*
*Programming Languages, Genuys, F. (Editeur),*
Academic Press, Londres, 1968, pp. 113-289.

[5] Bacchus, P., Drieux, B.,
*Extension de la forme normale de Backus pour la détection d'erreurs syntaxiques,*
Actes du 5ème congrès de l'AFIRO, 1966, pp. 335-337.

[6] Groupe Algol 68 de l'AFCET, Bacchus, P., Pair, C., Peccoud, D. (Editeurs), *Manuel pratique de programmation ALGOL 68,*
à paraître chez Hermann, Paris.

[7] Bazerque, G.,
*Reconnaissance des identificateurs dans les textes ayant une structure de blocs,*
RIRO, B.3, 1969, pp. 53-117.

[8] Branquart, P., Lewi, J.,
*A scheme of storage allocation and garbage collection for ALGOL 68,*
Report R133, MBLE Research Lab., Bruxelles, Juillet 1970.

[9] Branquart, P., Lewi, J.,
*On the implementation of Local Names in ALGOL 68,*
Report R121, MBLE Research Lab., Bruxelles, Septembre 1970.