Etude d'éléments fondamentaux des langages de programmation : contrôle de l'utilisation des objets et primitives d'exécution

Didier Bert

N° d'ordre

THESE

présentée à

L'Université Scientifique et Médicale de Grenoble

pour obtenir

Le grade de Docteur de troisième cycle

"Informatique"

par

DIDIER BERT

ETUDE D'ELEMENTS FONDAMENTAUX DES LANGAGES DE PROGRAMMATION:

CONTROLE DE L'UTILISATION DES OBJETS ET PRIMITIVES D'EXECUTION

Soutenue le 2 mai 1973 devant la commission d'examen

Messieurs	**N. GASTINEL**	**Président**
	M. GRIFFITHS	**Rapporteur**
	G. VEILLON	**Examinateur**
	P. JORRAND	**Invité**

UNIVERSITE SCIENTIFIQUE

ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

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

PROFESSEURS TITULAIRES

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

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

PROFESSEURS ASSOCIES

MM.	BULLEMER Bernhard	Physique
	RADHAKRISHNA Pidatala	Thermodynamique

PROFESSEURS SANS CHAIRE

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

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

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

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

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

MAITRES DE CONFERENCES ASSOCIES

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

CHARGES DE FONCTIONS DE MATIRES DE CONFERENCES

Mme BERIEL Hélène	Physiologie
Mme RENAUDET Jacqueline	Microbiologie

Fait le 8 MARS 1972.

Un travail de recherche n'est pas une oeuvre spontanée, mais le long murissement de quelques idées qui se forment grâce à la confrontation des avis de plusieurs personnes. C'est ainsi que je voudrais citer et remercier ceux qui, de près ou de loin, ont contribué à l'aboutissement de ce travail :

- *Monsieur GASTINEL qui a bien voulu présider le jury de thèse et avec qui j'ai eu des discussions constructives ;*

- *Monsieur GRIFFITHS qui a su me laisser une certaine autonomie dans mon travail tout en s'intéressant de très près aux résultats que l'on pouvai en attendre ;*

- *Monsieur JORRAND qui a été à l'origine de nombreuses idées contenues dan ma thèse et dont la grande expérience en matière de langages de programmation m'a été d'un précieux secours tout au long des mois pendant lesqu nous avons travaillé ensemble ;*

- *Monsieur VEILLON qui a bien voulu juger mon travail avec beaucoup d'attention ;*

- *mes collègues de bureau, notamment P. COUSOT et Ph. CHATELIN qui m'ont aidé à résoudre quelques problèmes techniques.*

Enfin, je tiens à remercier les personnes des services de dactylographie et de tirage qui ont réalisé cette thèse.

Didier BERT

TABLE DES MATIERES

INTRODUCTION

1 - PRESENTATION GENERALE DU SYSTEME ---------- 1
1.1 - L'extensibilité des langages ---------- 1
1.2 - Eléments constitutifs d'un langage ---------- 2
1.3 - Insertion des extensions ---------- 4

2 - DEFINITION DES OBJETS DE BASE ---------- 7
2.1 - Structure générale ---------- 7
2.1.1 - Schéma du traitement d'un programme ---------- 8
2.1.2 - Définition des objets ---------- 10
2.2 - Objets du langage ---------- 12
2.2.1 - Objets de type données ---------- 13
2.2.2 - Principales opérations de base ---------- 17
2.2.3 - Objets de type programme ---------- 18
2.2.3.1 - Les modules ---------- 18
2.2.3.2 - Actualisation des modules ---------- 20
2.2.3.3 - Les locus ---------- 21
2.2.3.4 - Liaison des paramètres ---------- 24
2.2.3.5 - Désactualisation ---------- 25
2.2.3.6 - Opérateurs ---------- 25
2.2.3.7 - Interpréteur ---------- 26
2.2.3.8 - Extension de l'interpréteur ---------- 26

3 - CONTROLE D'EXECUTION ---------- 30
3.0 - Introduction ---------- 30
3.1 - Principe d'exécution ---------- 30
3.2 - Moniteur ---------- 31
3.3 - Fonctions de contrôle ---------- 32
3.3.1 - Primitives de contrôle ---------- 32
3.3.2 - Primitives de synchronisation ---------- 35
3.3.3 - Fonctions de composition ---------- 37
3.4 - Expressions de contrôle sans parallélisme ---------- 37
3.4.1 - Exécution séquentielle ---------- 38
3.4.2 - Exécution collatérale ---------- 38
3.4.3 - Branchements ---------- 38
3.4.4 - Groupes itératifs ---------- 39

3.4.5 - Appels de sous-programmes ---------- 40
3.4.6 - Appels récursifs ---------- 41
3.4.7 - Coroutines ---------- 43
3.4.8 - Structure de bloc ---------- 45
3.4.9 - Réentrance ---------- 47
3.4.10 - Marche arrière ---------- 47
3.4.11 - Branches de programme indexées ---------- 48
3.5 - Problèmes de parallélisme ---------- 50
3.5.1 - Interruptions ---------- 50
3.5.2 - Synchronisation de processus ---------- 51
3.5.2.1 - Section critique non partageable ------- 52
3.5.2.2 - Opérations P et V de Dijkstra ---------- 53
3.5.2.3 - Synchronisation ---------- 54
3.5.2.4 - Exécution asynchrone ---------- 55

4 - CONTROLE DES TYPES A LA COMPILATION - MECANISME DES CLASSES - 57
4.1 - Les types dans les langages de programmation --------- 57
4.2 - Les classes ---------- 59
4.2.1 - Création d'une classe ---------- 59
4.2.2 - Opérations d'ensemble sur les classes --------- 60
4.2.3 - Graphe des classes ---------- 61
4.2.4 - Classes vides ---------- 62
4.2.5 - Produit cartésien ---------- 63
4.3 - Relations fonctionnelles ---------- 64
4.3.1 - Les conversions ---------- 64
4.3.2 - Les niveaux ---------- 65
4.3.3 - Les procédures ---------- 66
4.4 - Appartenance à une classe ---------- 68
4.5 - Utilisation des graphes ---------- 68
4.5.1 - Principe du contrôle des classes ---------- 68
4.5.2 - Opération sur les niveaux ---------- 69
4.5.3 - Opération sur les relations ---------- 70
4.5.4 - Application des conversions ---------- 72

5 - DEFINITION FORMELLE ---------- 74
5.1 - Présentation de la définition formelle ---------- 74
5.2 - Règles de métasyntaxe ---------- 76
5.3 - Définitions préalables pour l'interprétation --------- 76
5.4 - Structure du programme ---------- 78
5.5 - Evaluation des unités ---------- 82

5.6 - Traitement des identificateurs ---------- 83
5.7 - Objets de base ---------- 85
5.8 - Objets de type programme ---------- 90
5.9 - Objets utilisés dans le contrôle d'exécution --------- 95
5.10 - Contrôle d'exécution ---------- 95

6 - EXEMPLES D'EXTENSIONS ---------- 103
6.1 - Extensions pour l'allocation des variables ---------- 103
6.2 - Extension par actualisation ---------- 111
6.3 - Dialecte à structure de bloc ---------- 117
6.4 - Exemple d'utilisation des coroutines ---------- 124
6.5 - Dialecte de programmes non déterministes ---------- 135
6.6 - Langage classifié ---------- 149

CONCLUSION ---------- 156

Annexe 1 : OPERATEURS DE BASE ---------- 159
Annexe 2 : SYNTAXE CONCRETE DU LANGAGE DE BASE ---------- 166
Annexe 3 : ARTICULATION SOFTWARE - HARDWARE ---------- 170
Annexe 4 : FORMALISME DES MACROS-SYNTAXIQUES ---------- 173

BIBLIOGRAPHIE

INTRODUCTION

Les langages de programmation prolifèrent au point que l'on a pu parler de "tour de Babel" de l'âge moderne. C'est pourquoi je n'ai pas l'intention de définir un nouveau langage pour les utilisateurs qui entrerait en concurrence avec les langages déjà existant, mais de mettre en lumière certaines idées relatives aux langages de programmation. Pourtant ce travail se présente comme la définition d'un langage ; cela tient essentiellement à trois choses :

- le souci de rassembler les différentes idées dans un tout organique et de remettre chaque élément à sa place ;
- le côté "historique" de mon travail de recherche qui consistait à l'origine à définir un langage de base dans un système de programmation extensible avec MM. JORRAND (Centre Scientifique I. B. M.), SCHUMAN (Centre Scientifique I. B. M.) et COUSOT (IRMA-Grenoble) ;
- le sentiment que l'on peut mieux se rendre compte de l'intérêt de certaines innovations lorsqu'il est possible de les utiliser pour écrire des algorithmes et dans une étape ultérieure, de les tester directement sur ordinateur.

Ce langage de base est décrit d'une façon très simple avec une syntaxe rudimentaire, mais on ne s'est pas préoccupé de la forme extérieure puisque celle-ci est aisément modifiable par un macro-processeur et des extensions syntaxiques [30] [27] [36], d'ailleurs prévus dans le projet initial. Le chapitre 1 présente les différents éléments qui semblent fondamentaux et dont chacun constitue un tout logique ; les chapitres 2, 3, 4 reprennent chacune de ces divisions, à savoir :

- définition des objets : données et traitement de ces données ;
- définition du contrôle d'exécution dans l'évaluation des programmes
- définition du contrôle statique sur la représentation des données et leur utilisation.

Les deux premiers points constituent ce que l'on pourrait appeler un langage "sans type" et le troisième montre comment l'on peut introduire le mécanisme des types ou modes au niveau de la compilation, ainsi que les conversions associées. Parallèlement aux chapitres 2 et 3 qui sont des descriptions informelles de la partie langage de base, le chapitre 5 donne la définition formelle, à la fois syntaxique par la donnée de la syntaxe concrète, et sémantique, par la syntaxe abstraite et l'interpréteur associé. Divers types d'extensions sont donnés au chapitre 6 utilisant les mécanismes internes d'extension, et les macros pour clarifier l'écriture.

Les innovations par rapport aux langages traditionnels sont articulées autour de trois points fondamentaux.

(1) D'une part, le contrôle précis par le programmeur de la gestion de ses "modules" (2.1) par l'allocation explicite des variables ("actualisation" 2.2.3.2) et un repérage des modules alloués par des pointeurs appelés "locus" (2.2.3.3). Cela a pour conséquence la possibilité d'allouer plusieurs fois le même module avec éventuellement des paramètres différents et d'y avoir accès simultanément. Cette idée généralise des notions bien connues — les procédures récursives font des allocations à chaque appel, mais on n'a accès qu'à la dernière "actualisation" — et ouvre la porte aux nouveautés introduites par les systèmes à multitraitement : simultanéité, parallélisme, etc....

(2) Le deuxième point, qui a obligé entre-autres à introduire l'actualisation, est le contrôle du flot d'exécution par les "index" (ch.3). L'exécution d'un programme n'est plus automatique ; elle doit être commandée par l'activation d'un index qui contient toutes les informations relatives à l'évaluation d'un "module". Les "fonctions de contrôle" qui agissent sur les index produisent le déroulement de l'exécution : enchainement des instructions, mises en attente, traitement des interruptions, etc.

(3) Le troisième point qui est un peu isolé des deux précédents est un mécanisme d'introduction automatique de conversions entre des "classes" d'objets, ces classes étant définies en fonction de leur utilisation (ch.4). C'est un problème qui se rattache à la notion des types ou des modes des langages de programmation, tout en ayant quand même de grandes particularités.

Après avoir brossé les lignes de force de ce travail, il ne reste plus qu'a entrer dans les détails plus ou moins techniques de la définition du langage, en se donnant rendez-vous à la conclusion pour ouvrir cette étude sur les directions vers lesquelles elle peut nous entraîner.

CHAPITRE 1

1. PRESENTATION GENERALE DU SYSTEME

1.1. L'extensibilité des langages

Un sujet aussi vaste que celui de l'extensibilité des langages ne peut être traité dans un paragraphe; aussi nous nous contenterons de raisonner à l'aide de quelques points particuliers.

La première notion d'extension est celle de la définition de sous-programmes (Fortran : "Subroutines" et "functions"). L'appel des sous-programmes met uniquement en jeu le passage de contrôle après liaison des paramètres. Ensuite sont apparues, pour les langages de bas niveau (assembleur), les macros qui sont aussi une forme d'extension, mais où l'utilisation consiste en une recopie du corps de la macro avec substitution des paramètres. L'idée des macros a beaucoup évolué avec la notion de transformation de chaînes, puis de transformation d'arbres syntaxiques survenant aux diverses étapes du processus de compilation [9] [40]. Dans les langages extensibles proprement dits, un grand pas a été fait dans le problème de l'extensibilité des données, avec la possibilité de construire de nouveaux modes [10] [45] [47] [41] et corrélativement, d'introduire des opérateurs sur ces nouveaux types de données, opérateurs qui ne sont, en fait, que des appels de sous-programmes avec une commodité d'écriture ressemblant à une transformation du type macro [23] [19]. Il est bien connu que cette extension des opérateurs peut ne pas être efficace puisqu'elle engendre une série d'appels en cascades qui est souvent relativement lourde [42]. D'où l'idée récente de supprimer ces appels par une meilleure définition des opérateurs et si possible une optimisation du code qui en fasse en quelque sorte des "fonctions incorporées" (interpréteur extensible [37]). Enfin, un problème qui reste

encore en partie ouvert est celui de la manipulation du contrôle d'exécution et celui - dépendant - de l'environnement d'exécution. Il semble que, dans un système complet, on puisse simuler des formes non habituelles du passage de contrôle (coroutines, "marche arrière", multi-environnement) [34], mais ces formes ne sont pas introduites dans les langages car elles amènent le rejet de la notion d'allocation classique de variable et l'adoption de méthodes plus ou moins complexes [6]. Evidemment, il est intéressant de se demander, par exemple, quelles sont les formes de contrôle d'exécution qui autorisent l'allocation des variables en pile et dans ce sous-ensemble, on obtiendrait l'appel récursif de procédures, la structure de bloc, mais aussi peut-être d'autres possibilités que l'on n'a pas l'habitude d'utiliser. Nous n'avons malheureusement pas eu le temps de travailler à ce problème théorique, mais les idées dégagées au chapitre 3 peuvent donner des bases de départ.

1.2. Eléments constitutifs d'un langage

Afin d'essayer de voir plus clair dans la façon dont on peut concevoir un langage de programmation, nous avons essayé, Ph. Jorrand et moi-même, de faire une analyse de ses éléments constitutifs. Il semble que l'on puisse distinguer une séparation assez nette entre d'une part, un langage de base proprement dit où les données sont décrites en terme de représentation interne et où les opérateurs se distinguent en fonction du codage de ces données (addition en fixe, addition en flottant ...) et d'autre part, un ensemble de types - ou modes - indépendants ou structurés (entier, caractère, tableau ...) qui sont des entités que le programmeur peut manipuler d'une façon externe et, parallèlement, les procédures génériques et les conversions automatiques entre les types.

Le langage de base sans type peut lui-même être décomposé en deux parties d'importance variable suivant les langages :

1) Définition des objets et expressions sur ces objets.

2) Fonctions du contrôle d'exécution.

Les langages habituels mélangent ces deux groupes d'actions :

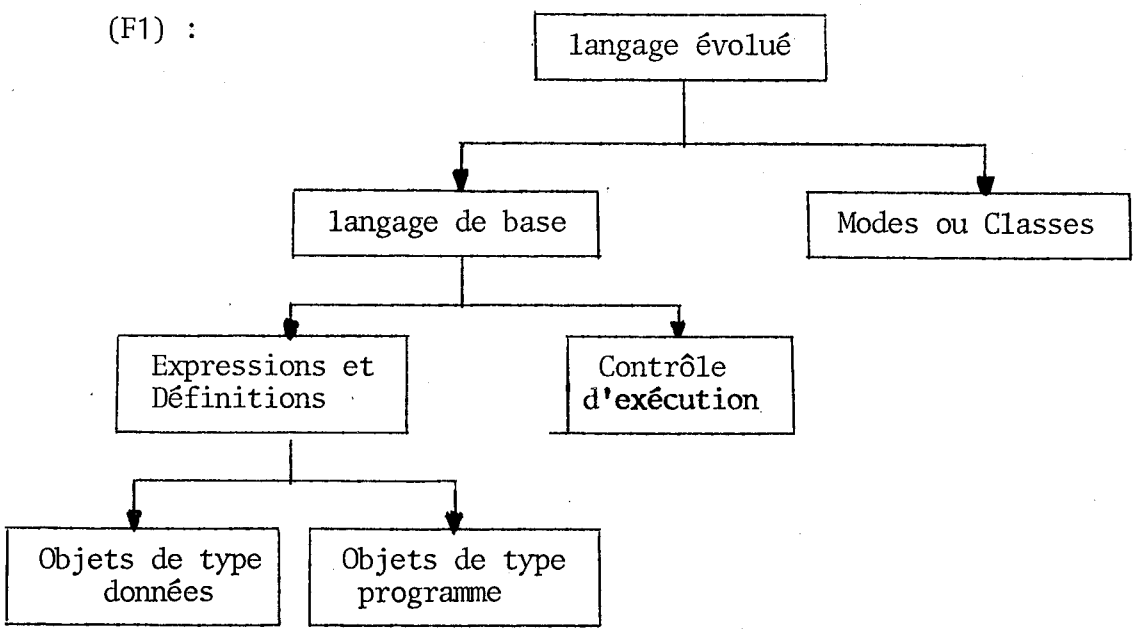
(Algol) x := y+4; goto L;

Dans Lisp, il n'y a pas de partie (2) - sauf dans la forme "programme"-.

Parmi les objets du langage de base, il y a bien sûr la distinction élémentaire entre objets de type données et objets de type programme. Ces derniers ont la propriété d'être exécutables, c'est-à-dire qu'ils représentent une action, et doivent être traités d'une façon spéciale.

Pour schématiser, on a la figure (F1) où les flèches représentent la décomposition de l'entité supérieure en entités plus simples.

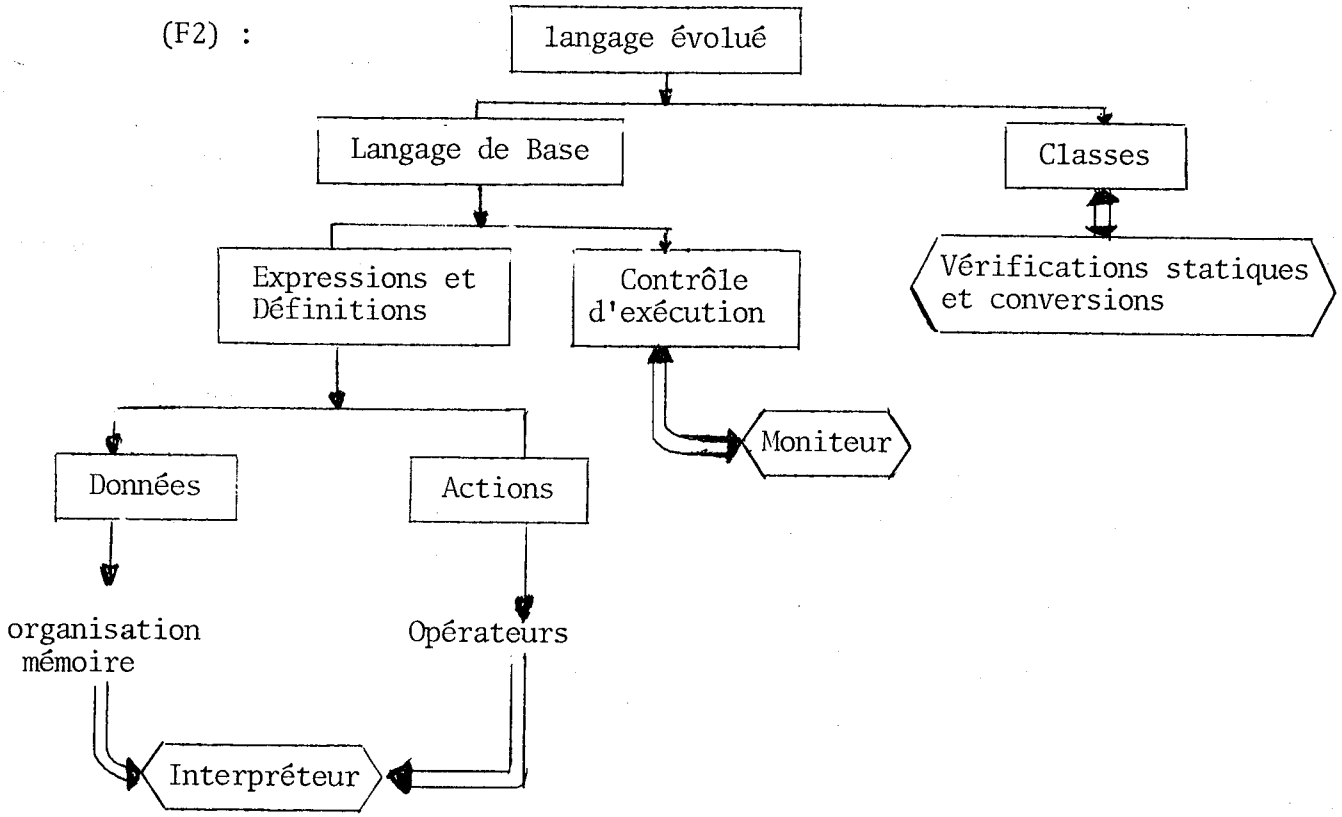
(F1) :



Derrière chacun de ces éléments, on trouve une partie du compilateur ou de la "machine" qui exécute un tel programme. Avec le mécanisme des types ou modes - que l'on a redéfini ici en l'appelant mécanisme des classes, on trouve les vérifications statiques et compilation de

conversions; parmi les objets de type programme, il y a les opérateurs qui sont évalués à l'aide d'un interpréteur; enfin, les fonctions de contrôle sont exécutées à l'aide d'un moniteur. On retrouve au niveau hardware les équivalents du moniteur dans l'unité de contrôle et de l'interpréteur dans l'unité arithmétique et logique, mais les éléments hardware n'étant pas extensibles ni adaptables, le problème de compilation est évidemment de réaliser les interfaces. Cela nous donne le schéma modifié où l'on a noté avec une flèche double la liaison avec les organes d'exécution.

(F2) :



1.3. Insertion des extensions

Sur le schéma (F2) on peut placer les extensions en les diversifiant suivant la branche sur laquelle ils s'appliquent :

1) Le mécanisme des classes est un ensemble qui comprend son propre système d'extension et s'appuie sur des définitions qui génèrent un graphe de conversions et de procédures (chapitre 4).

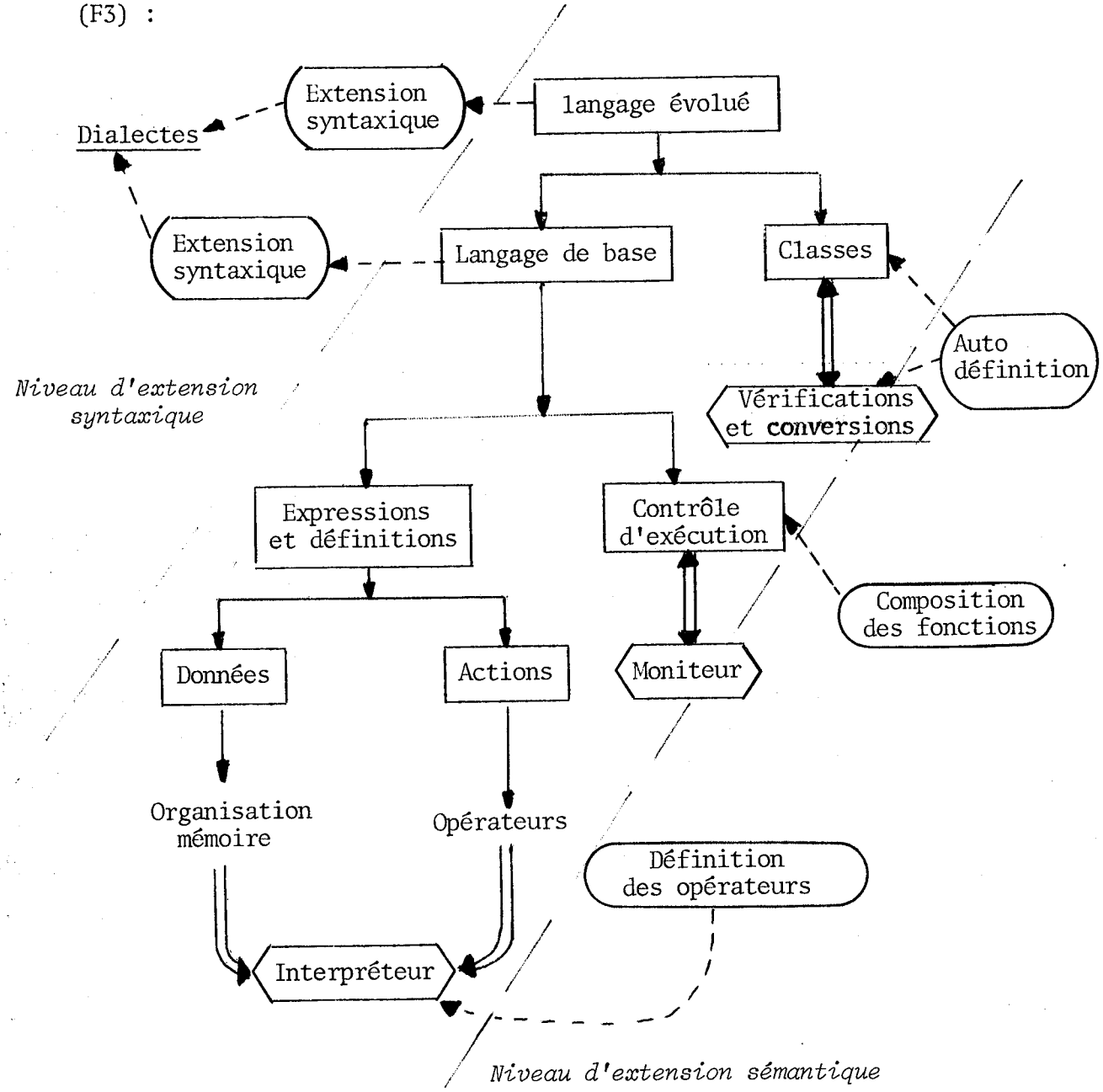
2) Les fonctions de contrôle peuvent se combiner à l'aide de deux constructeurs pour former des fonctions composées adaptées à l'actic à entreprendre (chapitre 3).

3) La définition des opérateurs est une "compilation" qui ir serre de nouvelles entrées à l'interpréteur ce qui le rend extensible (§ 2.2.3.9.).

Ces trois types d'extensions sont intrinsèques à la définition du langage et on peut les appeler des extensions de niveau 1 ou exte sions sémantiques. Par contre, on peut, soit à partir du langage de base seul, soit à partir du langage évolué introduire un mécanisme d'extensior genre macros qui a pour but de simplifier l'écriture des expressions de base et produire des "dialectes" simples mais diversifiés suivant le besoin de l'utilisateur. Ce sont les extensions de niveau 2 ou extensions s taxiques; elles ne sont pas étudiées ici, mais nous nous en servirons pou les exemples de la partie 6.

Le schéma du système complet muni de ses extensions est :

(F3) :


Extension syntaxique
langage évolué
Dialectes
Extension syntaxique
Langage de base
Classes
Auto définition
Niveau d'extension syntaxique
Vérifications et conversions
Expressions et définitions
Contrôle d'exécution
Composition des fonctions
Données
Actions
Moniteur
Organisation mémoire
Opérateurs
Définition des opérateurs
Interpréteur
Niveau d'extension sémantique

CHAPITRE 2

2. DEFINITION DES OBJETS DE BASE

2.1. Structure générale

L'élément fondamental du langage est le module qui contient un ensemble d'informations, à la fois des définitions et des "calculs", c'est-à-dire les données et leur traitement. Un module s'écrit :

mod (declass (...), (I_1, I_2, ..., I_n), u_1 u_2 ... u_k)

où declass (...) définit le mécanisme des classes du module (cf. ch. 4),

(I_1,...,I_n) est la liste d'index nécessaires à l'exécution (cf. Contrôle d'exécution - ch. 3),

u_1 u_2 ... u_k est l'ensemble des "unités" de programme.

L'unité de programme que l'on appellera simplement par la suite unité est composée de trois éléments entre parenthèse et séparés par des virgules :

(étiquette, expression, contrôle)

où

- "étiquette" nomme le point d'emplacement de l'unité;
- "expression" est un ensemble d'opérations de base;
- "contrôle" est la partie de contrôle d'exécution en liaison avec les index.

La dichotomie dont on a parlé au chapître précédent entre les opérations sur la mémoire et le contrôle ou enchaînement d'exécution se retrouve au niveau de l'écriture de l'unité ainsi qu'au niveau des "organes" d'exécution :

. un interpréteur (2.2.3.7) exécute les opérations sur les données;

. un moniteur (3.2) exécute les opérations de contrôle.

L'expression peut être, soit une définition,

def (identificateur, expression de définition)

soit un appel d'opérateur (2.2.3.6)

plus (A, B)

soit une combinaison de définitions et/ou d'opérateurs séparés par des virgules.

Dans ce dernier cas, les expressions simples sont exécutées dans un ordre quelconque puisque la notion d'ordre d'exécution n'est explicite qu'au niveau du contrôle.

2.1.1. Schéma du traitement d'un programme

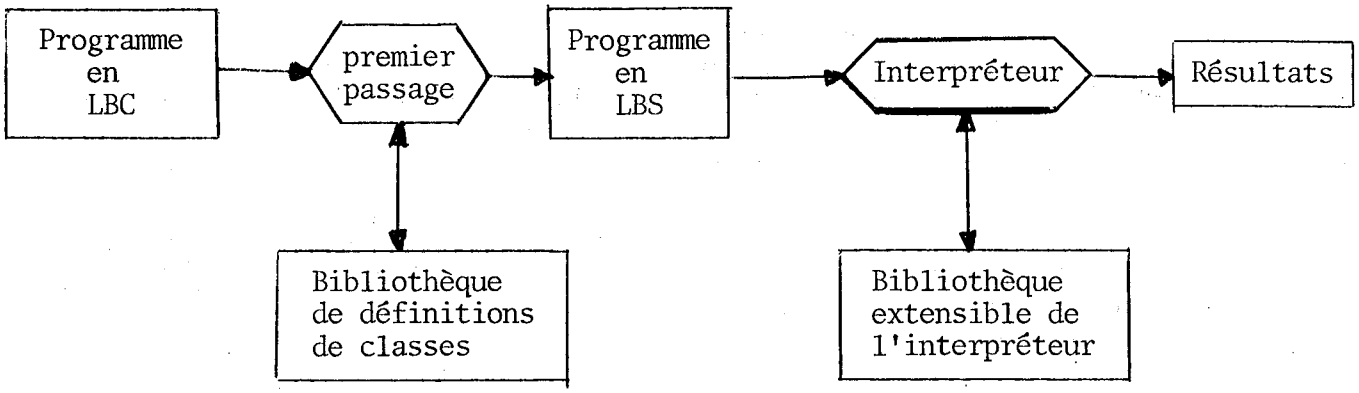
Il y a deux sortes de définitions :

- les définitions d'objets de base (2.1.2) qui sont contenues dans les unitées;
- les définitions du mécanisme des classes (4.2) qui se trouvent dans le premier champ de la définition d'un module, sous le sélecteur declass.

Les premières sont élaborées lors de l'interprétation (ou de l'exécution); les deuxièmes sont élaborées lors du premier passage (ou compilation).

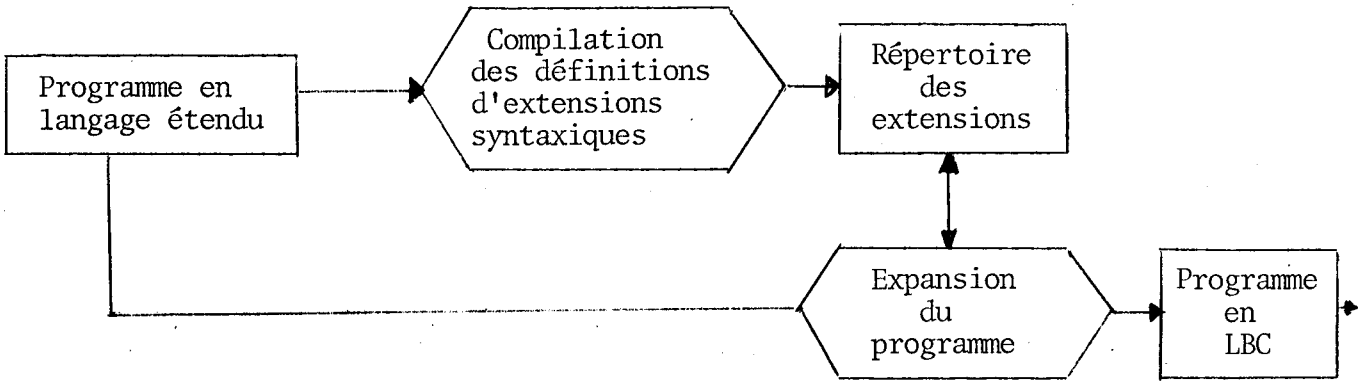
D'où le schéma F4 dans lequel on distingue le langage de base classifié : LBC et le langage de base simple : LBS. Les flèches horizontales représentent le traitement du programme donné, en fonction des informations des bibliothèques :

(F4)



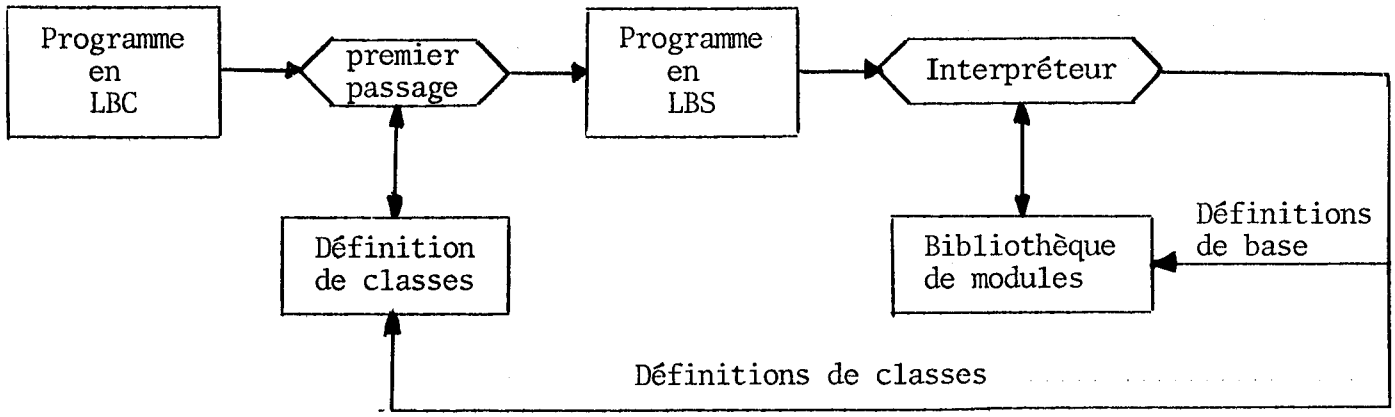
Le programme en LBC peut être produit à partir de définitions de macros, d'où la chaîne initiale :

(F5)



Le programme peut consister uniquement en définition de classes et de modules. Dans ce cas, le résultat n'est pas l'exécution d'un algorithme, mais la mise à jour des bibliothèques des classes et des modules en vue d'une exécution ultérieure. Le schéma d'extension des bibliothèques est :

(F6)



Un programme consiste donc en définition de modules :

def (M1, mod (—))

def (M2, mod (—))

ou en exécution de modules :

execute (index, module exécutable)

Les "modules exécutables" seront définis en 2.2.3.2.

2.1.2. Définition des objets

Un objet O est un élément du langage représentant une valeur et pouvant être identifié. La liste des objets du langage est donnée en 2.2. La définition d'un objet se fait par l'opérateur def :

def (identificateur, objet)

Une définition est élaborée à l'exécution. L'objet qui est habituellement une expression est évalué et sa valeur est associée à l'identificateur pour la suite de l'exécution, à la manière des déclarations d'identité d'Algol 68 [45]. Il n'y a pas de restrictions syntaxiques dans la position

des définitions, mais elles ne peuvent être calculées qu'une fois dans un module donné, de même qu'un identificateur ne peut être défini qu'une fois, excepté pour les procédures et les conversions (cf. traitement des classes, § 4.3.).

Les identificateurs ont la portée du module dans lequel ils sont définis. Il y a cependant deux cas particuliers à cette règle :

(1) **Modules internes** : Les modules internes peuvent être définis dans un module :

def(M,mod(... def(M1,mod( )) ...))

M1 est un module interne à M. A l'intérieur du module M1, sont valides :

- les identificateurs définis en M1;
- les identificateurs définis dans un module englobant et non redéfinis à l'intérieur de M1 (principe des blocs Algol 60 [32]).

(2) **Modules externes** : Les identificateurs définis dans les modules externes ne sont pas connus puisque leur portée est locale. Néanmoins, les noms des modules sont connus dans la bibliothèque. Si l'on veut utiliser un identificateur dans un module externe, on fait référence à ce module par la notation use :

use (M1)

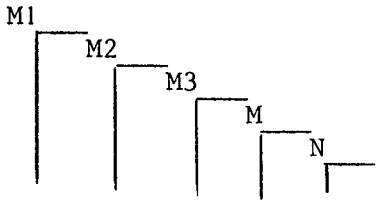
où M1 est un module défini antérieurement. L'identificateur non défini dans le module où il apparaît ni dans les modules englobants sera recherché dans le module M1. Le use peut avoir en paramètre une liste de modules :

use(M1, M2, ..., M_k)

La recherche de l'identificateur se fait alors en commençant par le dernier. On génère donc une nouvelle structure de blocs :

def(M, mod (... def(N, mod (... use (M1,M2,M3)...))...))

Dans M on ne reconnaît que les identificateurs locaux; dans N au **contraire**, on trouve l'imbrication :



Il est à noter que l'utilisation d'un identificateur implique que la définition de cet identificateur ait été élaborée et donc que le module auquel on fait référence par le use ait été exécuté. Ces notions seront reprises lors des opérations sur les modules (2.2.3.2.).

La portée du use est statique et s'applique à tout le module. Il est indispensable si l'on fait la compilation des modules; si on interprête entièrement le programme, la recherche dynamique des identificateurs permet de résoudre les références externes. On reparlera de ce problème des références externes avec l'introduction de la notation '.' (2.2.3.3.).

2.2. Objets du langage

Le langage de base manipule les objets, donc les types de valeur, suivant, dits de type 1 :

- les références,
- les adresses,
- les déplacements,
- les constantes,
- les modules,
- les opérateurs,
- les locus ou lieu d'action,

tous ces objets sont manipulés dans la partie expression des unitées. Deux autres objets sont utilisés dans les fonctions de contrôle; il s'agit de ceux de type 2 :

- les index,
- les signaux.

Les objets de type 1 sont considérés comme des valeurs et à ce titre, ont une représentation bien précise en mémoire, peuvent être utilisés dans les expressions, c'est-à-dire servir d'opérandes, être des résultats d'expressions et être affectés à des références. Ils satisfont au principe de complétude donné par Reynolds [35] (voir aussi POP-2 [7]).

2.2.1. Objets de type données

(1) Les références. Elles représentent des zones de mémoire. Leur expression littérale est :

ref (adresse, taille)

La zone de mémoire définie par ce ref commence à l'adresse désignée par le premier champ et couvre l'emplacement indiqué par le deuxième champ.

(2) Les adresses. Ce sont des points dans une zone de mémoire. Leur expression est :

adr (référence, déplacement)

où la référence est la zone dans laquelle se situe l'adresse, et le deuxième champ indique la valeur de déplacement entre l'origine de la référence et le point de cette adresse.

(3) Les tailles ou déplacements. Très généralement, on peut dire qu'ils permettent de donner une "mesure" dans l'espace des adresses :

dep(expression de déplacement)

Mais là se pose un problème que nous retrouverons et dont nous faisons l'inventaire en annexe 3. Alors que l'introduction des modes permet de rendre un langage "indépendant de la machine", un langage comme celui-ci ne peut pas ignorer les caractéristiques physiques, si l'on veut que le langage reflète les possibilités de la machine. C'est ainsi que la règle donnant le déplacement peut être modifiée au gré de l'implémenteur :

- s'il n'y a qu'un seul type de zones dans la mémoire physique (caractère, mot, ...), il suffit d'en indiquer le nombre :

  dep (3)

- si par contre il y a plusieurs types de zones et plusieurs types d'adressage, il faut préciser ce que j'appelle le "déplacement unitaire" qui sert aussi à effectuer le cadrage des opérations :

  dep (3, byte)

Pour la suite de mon travail, j'ai pris comme cas particulier le système d'adressage du 360 IBM avec les conventions suivantes :

dep (expression, déplacement unitaire)

où - expression doit donner un résultat entier,
- déplacement unitaire est une chaîne de caractères (cf. constantes) qui est l'une des chaînes suivantes :

B	pour bit
BYTE	pour byte ou octet
HW	pour demi-mot (2 octets)
W	pour mot
DW	pour double-mot.

L'imbrication des références et des adresses fait qu'une référence doit être contenue dans la zone spécifiée dans les champs de son adresse. De ce fait, on a besoin de définir une référence globale qui contiendra toutes les autres. Ceci est fait par une fonction mem qui retourne une zone de mémoire centrale allouée par le système. Cette fonction a un paramètre qui est la taille de la zone :

def(DIXMOTS, mem (dep (5, car(DW))))

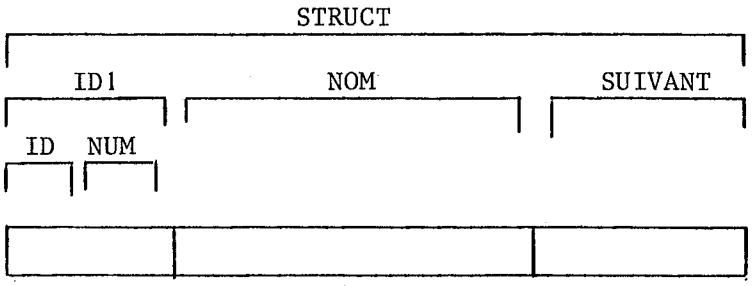
L'allocation des variables peut être gérée par l'utilisateur dans la zone ainsi définie, ou laissée au système par des emplois successifs de mem.

EXEMPLES

(E1) - Définition d'une référence d'un mot à partir d'une adresse AD1 :

def (REFERENCE, ref (AD1, dep (1, car (W))))

(E2) - Définition d'une structure comportant les champs :



On écrit en langage de base strict :

```
def (STRUCT, mem (dep (4, car (W))))
def (ID1, ref (orig (STRUCT), dep (1, car (W))))
def (ID, ref (orig (ID1), dep (1, car (BYTE))))
def (NUM, ref (adr (ID1, dep (1, car (BYTE))), dep (3, car (BYTE))))
def (NOM, ref (adr (STRUCT, dep (1, car (W))), dep (8, car (BYTE))))
def (SUIVANT, ref (adr (STRUCT, dep (3, car (W))), dep (1, car (W))))
```

Il est évident qu'une telle écriture est extrêmement lourde, aussi il convient pour une implémentation donnée d'écrire des macros qui génèrent le texte ci-dessus - ou l'arbre d'analyse correspondant - à partir d'un texte en langage étendu plus simple à l'utilisateur (6.1.2.). Cette remarque est valable chaque fois qu'une forme se complique.

(E3) - Allocation des variables : un opérateur peut être écrit (6.1.1.), qui retourne une référence à partir de la première adresse libre dans la mémoire; il s'agit de l'opérateur taille que l'on utilise ainsi :

```
def(X, taille (1, car(W)))
def(Y, taille (3, car (BYTE)))
```

donc X est une référence à la première adresse libre dans une zone définie initialement et de taille 1 mot. Y est de même une référence de longueur 3 bytes.

(4) Les constantes. On retrouve avec les constantes le problème de la liaison avec la machine. Les constantes sont les valeurs sur lesquelles sont faites les opérations habituelles, mais cette valeur en machine est manipulée en fonction d'un certain codage interne. Evidemment on pourrait tout ramener à la notion de chaîne de bits, mais ce serait redéfinir avec des programmes ce que la machine fait d'une façon cablée. Il faut donc partir d'un niveau plus élevé mais ce niveau n'étant pas le même pour chaque machine, l'implémenteur doit définir les types de codage qui peuvent être acceptés. Dans le cas du 360 IBM, on a choisi le jeu de constantes suivant :

caractère : car (suite de caractères - sauf) -)
bit : bit (suite de 0-1)
hexadécimal : hexa (suite de 0-9 ou A-F)
nombre en fixe : fx (suite de 0-9)
nombre flottant : fl (suite de 0-9 avec point décimal, puissance de 10)
- Si la puissance est omise, elle est prise égale à 1.

On aurait pu utiliser la forme décimale (décimal "paqué") mais cela n'a pas été fait, dans le but de garder quand même une certaine généralité, et de plus, il est facile de l'ajouter dans une application particulière.

Une constante se présente donc comme un mot-clé suivi d'une chaîne à traduire en représentation interne. On peut admettre un cas particulier, c'est celui de fx qui peut être omis pour le codage en virgule fixe d'une chaîne de digits :

123 équivaut à fx(123)

Evidemment, une chaîne de digits intervenant dans une constante ne sera pas traduite en fixe :

car(123) n'équivaut pas à car(fx(123)).

On dispose aussi d'un certain nombre de fonctions (opérateurs) pour passer d'une représentation interne à une autre (cf. Annexe 1).

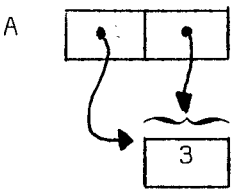
2.2.2. Principales opérations de base

On explicite ici quelques opérations élémentaires qui seront surtout utilisées dans les exemples :

val : prend un paramètre qui est une référence et donne comme résultat la valeur contenue dans cette référence.

store : prend deux paramètres : le premier est une référence, le second une valeur; il range la valeur à l'emplacement représenté par la référence.

store (A, 3)



Après cette instruction, val(A) donne 3

Sur les constantes de type fx on a les opérations arithmétiques définies par : plus, moins, mult, div, exp.

Comme la notation est préfixée, il n'y a pas de priorité sur ces opérateurs.

Sur les constantes de type car, on a les opérations de caractères :

conc pour concaténer deux chaînes :

conc(car (A B C), car(DE)) est la chaîne car (A B C D E)

sch pour prendre une sous-chaîne :

sch(car (A B C D E), 2, 3) est la chaîne car (B C D)

Sur les constantes de bits, on peut faire les opérations habituelles :

ou pour l'union

et pour l'intersection

ni pour le ou exclusif.

Relativement aux adresses et déplacement, on a les opérations :

addep (dep1, dep2) qui calcule le déplacement composé par ces deux déplacements avec les modifications dues au cadrage.

Une fonction sur les adresses peut être déduite de cette fonction :

compadr (ad1, dep2) qui est définie être équivalente à :

compadr (adr (ref1, dep1), dep2) ≡ adr (ref1, addep (dep1, dep2))

Etant donné une référence, la fonction orig donne l'adresse d'origine, tandis que pour une adresse, la fonction rf donne la référence :

orig (ref (ad1, dep1)) ≡ ad1
rf (adr (ref1, dep2)) ≡ ref1

L'ensemble des fonctions sur la mémoire est donnée en annexe 1.

La plupart des opérateurs ont un résultat qui est laissé sur la pile d'évaluation. Un opérateur met explicitement des objets au sommet de la pile :

stack(X)

A l'inverse :

decap (3) enlève les 3 éléments du sommet de pile.

2.2.3. Objets de type programme

2.2.3.1. Les modules

Les modules ont déjà été introduits en 2.1. Ils sont destinés à être exécutés par les index et renferment un ensemble d'informations à la fois définition, calcul et séquence d'exécution par le contrôle qui y est explicité. Lorsqu'ils sont définis, les modules ne sont pas encore exécutables. Ils peuvent contenir des objets "formels" pour la paramétrisation des calculs.

En vertu de la complétude du langage, tous les objets représentant une valeur peuvent être des objets formels; ainsi on a :

fref	référence formelle
fadr	adresse formelle
fdep	déplacement formel
fconst	valeur de constante formelle
fmod	module formel
fop	opérateur formel
floc	locus formel

Parmi les objets utilisés par le contrôle d'exécution, seuls les signaux peuvent être formels; les index peuvent être passés dans l'environnement d'exécution d'un autre module, mais ce passage est dynamique et ne nécessite pas les formels puisqu'il se situe au niveau de l'exécution.

fsig	est un signal formel

Un formel peut être utilisé partout où des objets de même type sont prévus :

- soit dans les définitions :

def (X, fref)

def (M, fmod)

- soit comme opérandes :

store (fref, fconst)

Pour rendre un module exécutable, il faut allouer ses variables et ses formels; c'est l'actualisation [12] ; cette opération est indispensable. Lorsqu'un module est actualisé, on peut lier les paramètres avec de nouvelles valeurs; c'est la liaison des paramètres.

2.2.3.2. Actualisation des modules

Le module formel est composé d'une part, du code des unités, d'autre part de plusieurs tables : table des index, table des formels, table des identificateurs (5.8). Ces tables donnent simplement les informations nécessaires à l'allocation. L'actualisation fait l'allocation des zones pour les formels et les variables locales à partir de ces tables. Elle donne comme résultat les adresses (internes) de la zone d'allocation des variables et de la partie code, avec mise à jour de certaines variables de l'actualisation. L'algorithme A1 décrit l'opération d'actualisation à l'aide des notations inspirées d'Algol 60. Les variables utilisées sont :

- CTRF : variable entière } dans le module formel ;
- PTRF : pointeur
- CTRD : variable entière dans la zone des données allouées;
- PTRAR : pointeur associé à la recopie du code.

A1 - Algorithme d'actualisation

La notation est : act(M,α,β,γ)
où M est l'identificateur de module formel;α,β,γ des expressions dont les résultats constituent les valeurs des paramètres.

(1) - Recherche du module formel M
Si CTRF = 0 alors { charger le code à l'adresse AD1 ;
PTRF := AD1 ;
PTRAR := adresse du module M ;
CTRF := CTRF + 1 ;
(2) - Allouer la zone des variables et des formels à AD2 ;
(3) - Evaluer les paramètres α,β,γ et initialisation des formels ;
(4) - CTRD := 1;
(5) - Mettre sur la pile d'évaluation une structure contenant les adresses (PTRF, AD2).

2.2.3.3. Les locus

Les locus ou lieu d'action ont été étudiés par Sintzoff [39] pour introduire des possibilités de coroutines dans les langages évolués, principalement Algol 68. Ils n'ont pas tout-à-fait le même sens que celui qu'ils ont ici, mais ont en commun le fait de donner une adresse d'une partie exécutable qui a été actualisée - c'est-à-dire, chez Sintzoff, déparamétrisée -. Leur principale différence vient du fait qu'ils ont certaines propriétés des index, comme celle de fournir un point d'entrée d'exécution. Ils sont destinés, dans notre langage à repérer un ensemble exécutable, c'est-à-dire qu'ils contiennent l'adresse du code et l'adresse des données, toutes deux fournies par l'actualisation. On définit un locus par :

def (L, locus)

On utilise un locus par la fonction :

alloc (L, act (M,α,β,γ))

qui charge le locus L par le résultat de l'actualisation de M.
On peut aussi allouer un locus à un autre :

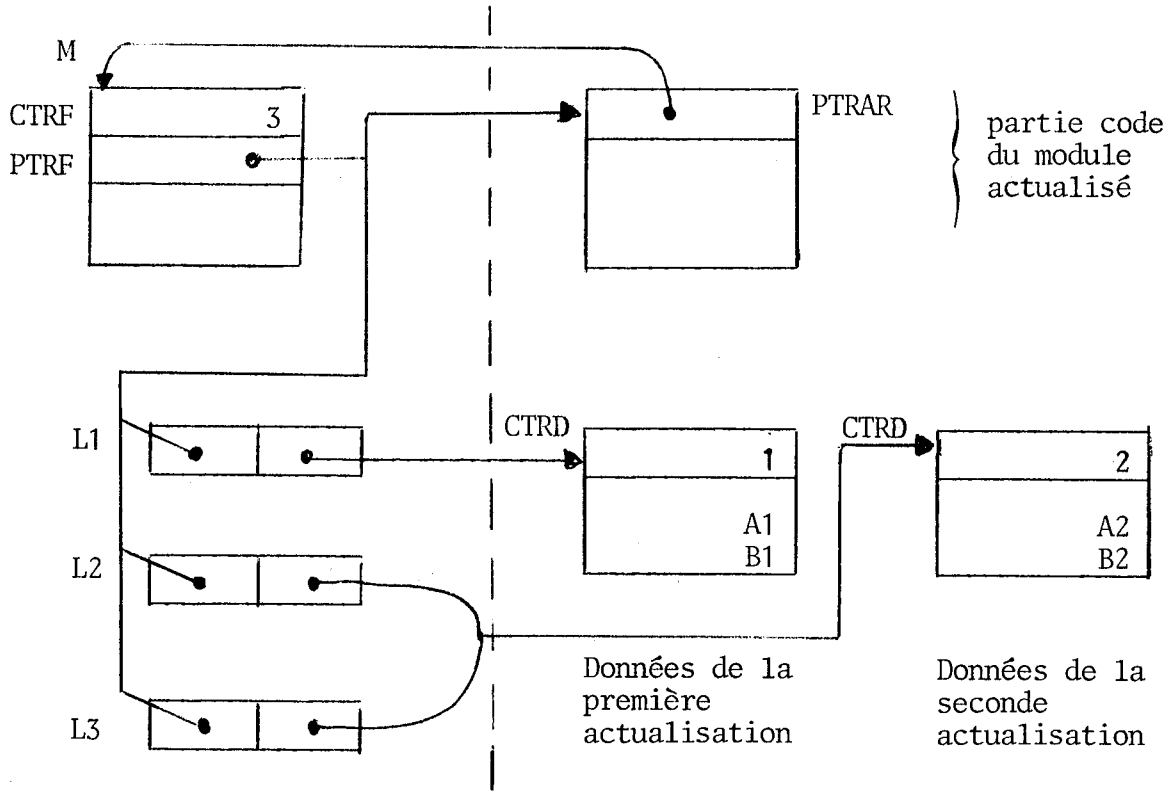
def (L1, locus)
alloc (L1, L)

L'allocation est différente de l'affectation (store) qui demande une référence comme premier argument.

En reprenant l'algorithme A1, on a traduit par un schéma (F7) les résultats de l'exemple suivant où M est un module formel à deux paramètres :

def (L1, locus), def (L2, locus), def (L3, locus)
def (M, mod (———))
alloc (L1, act (M, A1, B1))
alloc (L2, act (M, A2, B2))
alloc (L3, L2)

(F7)



Le compteur de la deuxième actualisation (CTRD) est à 2 à cause des deux locus L2 et L3 qui pointent sur cette zone de données.

D'un point de vue logique, l'actualisation fait le remplacement des paramètres - évalués - par recopie, comme Algol 68. Soient le module M1 :

```
def (M1, mod (, (I),
        (, def (X, fref), contrôle 1)
        (, store (X, plus (val (X), 1)), contrôle 2)))
```

et l'actualisation :

```
        def (L, locus)
        alloc (L, act (M1, A))    où A est une référencé
```

Le locus L repère un ensemble d'unités dont la sémantique est :

(, def (X, A), contrôle 1)

(, store (X, plus (val (X), 1)), contrôle 2)

Qualification dynamique des identificateurs externes

Soit l'exemple :

```
def (M, mod ( ——— def (X, ——— ) ———))
def (M1, mod ( ———  use (M) ——
                  :
               ———  plus (X, 1) ——— ))
```

Si X n'est pas défini dans M1, on prendra le X défini dans M d'après 2.1.2. Mais cette qualification n'est pas suffisante à cause des actualisations. En effet, complètons M1 par :

```
def (M1, mod ( ——— use (M)———
                  .
                  .
                  .
            (, (def(L1, locus), def (L2, locus)), ——— )
            (, (alloc(L1, act (M, -)), alloc (L2, act (M,-))), -)
               .
               .
               .
             ———————  plus (X,1) ———
```

Faut-il alors choisir pour X la définition de la zone de données pointée par L1 ou par L2 ? Il y a ambiguité. Celle-ci est levée par la notation :

locus . identificateur

c'est-à-dire ici :

——— plus (L1.X, 1) ———

Le use ne sert qu'à coder le X au moment de la compilation, en ce sens qu'il donne l'adresse relative dans la zone des données de M; cette adresse relative est basée par L1 qui donne à l'exécution l'origine de la zone des données à utiliser. Si on interprète le langage, le use n'est pas utile, puisque l'identificateur X peut être recherché à l'exécution à partir du locus L1 qui permet d'accèder à la table des identificateurs de M.

2.2.3.4. Liaison des paramètres

Le fait de l'allocation des variables locales lors de l'actualisation suppose que les différentes exécutions d'un module sont indépendantes. Si l'on veut, par exemple, écrire un module que l'on exécute chaque fois avec des paramètres différents, mais qui possède une variable locale comptabilisant le nombre d'exécutions, cela ne peut se faire simplement avec le mécanisme de l'actualisation; de plus, pour les cas courants, il n'est pas nécessaire de réallouer les variables à chaque exécution, d'où l'opération de liaison des paramètres. Cette opération est décrite par :

bind (locus, paramètres effectifs)

Le remplacement des paramètres effectifs lors de la liaisons - notamment leur nombre - dépend de l'actualisation. En effet, l'actualisation précise les paramètres qui pourront être modifiés ensuite par liaison. Ceci se fait de deux façons différentes :

(1) Option par défaut :

alloc (ℓ, act (MOD,α,,β,))

Le module MOD contient 4 objets formels; le premier et le troisième sont actualisés par les valeurs α et β et considérés comme objets internes dans l'actualisation; par contre, le deuxième et le quatrième pourront être liés par un bind. Le module actualisé par ℓ ne contient en définitive que deux formels. On peut écrire :

bind (ℓ, a, b)
bind (ℓ, a', b',)
etc...

(2) Option modified :

alloc (ℓ', act(MOD,α, modif(α1),β, modif(β1)))

Ici comme précédemment, α et β seront des variables locales à l'actualisation; par contre, α1 et β1 seront des valeurs initiales des deux paramètres pouvant être liés et donc modifiés ensuite par liaison :

bind(ℓ',a, b)
bind(ℓ', a', b') ...

2.2.3.5. Désactualisation

La désactualisation a essentiellement pour but de libérer la place prise par les zones mémoire d'une actualisation que l'on ne veut plus exécuter. Elle s'écrit :

desact (locus)

L'algorithme de désactualisation utilise les variables données en 2.2.3.2.

A2 - Algorithme de désactualisation

(1) Si CTRF > 1 alors CTRF := CTRF-1
sinon { libérer la partie code;
aller à (3)

(2) Si CTRD > 1 alors CTRD := CTRD-1
Sinon (3) libérer la zone des données.

2.2.3.6. Opérateurs

Les opérateurs forment la base des calculs dans un module. Ils constituent l'ensemble "fonctionnel" des expressions d'une unité, en ce sens que, étant donné certains arguments, l'opérateur fournit un résultat sans que l'on se soucie de l'algorithme qui a été utilisé pour y parvenir.

Exemple : Si l'on écrit factorielle (3), on ne s'intéresse pas à la façon dont est effectué le calcul. Par contre, si on écrit un module calculant factorielle pour des valeurs entières on est obligé de s'intéresser à l'algorithme - récursif ou itératif. C'est cette différence de point de vue qui existe entre les modules et les opérateurs (cf. option opmod).

La souplesse d'utilisation d'un langage tient surtout à l'introduction de nouveaux opérateurs. Il y a trois façons d'introduire les opérateurs, mais pour mieux comprendre ce mécanisme d'extensibilité, il convient d'abord de parler de l'appel des opérateurs et de l'interpréteur.

2.2.3.7. Interpréteur

Un interpréteur, selon une image classique [37], peut se représenter comme un aiguillage dans lequel chaque position correspond à un morceau de code qui est exécuté lorsque l'indicateur pointe sur cette position. En général, certains paramètres sont nécessaires lors de l'exétion; ces paramètres sont passés lors de l'appel. Un appel d'opérateur se fait donc par :

idop (A, B, C)

dans lequel :

- idop est la position de l'interpréteur à sélecter,
- A, B, C, les paramètres effectifs de l'exécution.

Pour rendre l'interpréteur extensible, il suffit d'une part d'indiquer un nom de point d'entrée - différent de ceux qui existent déjà - et d'autre part de donner le corps de la partie à exécuter. Il y a trois façons d'indiquer ce corps à exécuter, qui correspondent aux trois possibilités d'extension :

(1) - par introduction directe du code de l'opérateur.
(2) - par composition d'opérateurs existants.
(3) - par compilation d'un module.

2.2.3.8. Extension de l'interpréteur

La définition d'un nouvel opérateur se fait comme pour les autres objets par un def.

La première forme d'extension par introduction directe ne peut se faire qu'en respectant les conventions de liaison d'une implémentation donnée. Le corps de l'opérateur est donné en langage machine qui sera assemblé pour fournir le code exécutable. C'est ce type d'extension qui permet d'introduire les opérateurs de base :

def (store, op (corps de l'opérateur))

Dans le code il peut y avoir plusieurs alternatives dépendant des conditions de l'appel.

La deuxième forme permet de combiner des opérateurs entre eux avec optimisation de l'utilisation des registres. On écrit par exemple :

def (assign, opcomp (store (fref, val (fref))))

Ce qui signifie que l'on pourra utiliser assign de deux références :

assign (ref1, ref2)

avec l'effet que la valeur de la référence 2 est rangée à l'emplacement indiqué par la référence 1.

La troisième forme est la plus importante puisqu'elle permet de passer du module à l'opérateur, c'est-à-dire de la formulation "algorithmique" à la formulation "fonctionnelle" d'un calcul, et qu'elle effectue la compilation des modules.

Elle s'écrit :

def (id, opmod (locus, index, résultat))

"id" est toujours l'identificateur de l'opérateur;

"locus" est le module actualisé qui sert de source à la compilation;

"index" est l'identificateur d'index suivant lequel se fera l'exécution de l'opérateur, et donc la compilation (cf. 3.3.1.1.);

"résultat" est la liste des valeurs à mettre au sommet du stack après l'évaluation du module correspondant à l'opérateur. Ce troisième paramètre est :

stack (liste d'identificateurs du module).

La mise au sommet du stack permet d'utiliser les valeurs de résultat dans les expressions.

La sémantique d'un opérateur de module est simple à définir :

- soit la définition :

  def (id, opmod (L, I, stack (X, Y, Z)))

  où L repère une actualisation demandant deux paramètres :

  l'appel de l'opérateur :

  ——— id(A, B) ——

équivaut à :

—— bind(L, A, B),
exécution de L suivant l'index I,
stack(L.X, L.Y, L.Z) ——

Si l'on considère le module formel comme module source de la compilation, il y a analogie entre les paramètres de l'actualisation et les variables "gelées" de la compilation de ECL [20]. Le fait d'écrire :

def(id, opmod (act (M, A, modif (B)), I, stack (X)))

revient à dire de compiler M dans lequel A a été "gelée", et donc utiliser l'interpréteur pour évaluer les expressions dans lesquelles la variable A apparaît, avant la compilation. La spécification de l'index I sera éclaircie au chapitre 3, mais on peut dire déjà qu'il agit sur la compilation par la sélection de certaines unités du module, un peu à la manière des procédures génériques.

On a noté la différence d'écriture entre la définition d'un opérateur et son appel, qui se traduit ici par le soulignement. Elle représente le fait qu'un identificateur d'opérateur est rangé dans la table des définitions d'un module comme pour toute définition, mais que, une fois "évalué" (cf. 5.8), il représente une partie de code et un point d'entrée de l'interpréteur. Ainsi, lors de l'appel, l'action du soulignement signifie que l'on recherche l'opérateur directement dans l'interpréteur sans passer par la table des identificateurs. D'autre part, en mode purement interprétatif, il convient de souligner la similitude entre les opérateurs, primitifs (annexe1) ou définis, et les unités syntaxiques comme def, ref, etc... qui pourraient également être considérées comme opérateurs.

CHAPITRE 3

3. CONTROLE D'EXECUTION

3.0. Introduction

Les fonctions de contrôle dans les langages de programmation sont toujours relativement figées : goto, if then else, do while ... , appel de sous-programmes. Il n'y a pas d'unification ni de primitives de base. Il a semblé important dans le cadre de ce langage d'expérimenter un nouveau modèle du mécanisme de contrôle, assez simple pour être de base, mais assez général pour pouvoir englober toutes les possibilités connues et ouvert à toutes les nouveautés. Un séminaire a donné un aperçu du sujet [4], et avec l'aide de Ph. Jorrand, j'ai pu arriver à la formulation présentée ici. Si l'on veut être complètement maître de l'évaluation, il faut franchir une étape supplémentaire qui consiste à agir sur l'arbre des expressions pour lequel j'ai conservé un contrôle implicite qui est celui de l'interpréteur. Ces nouvelles idées sont consignées dans des notes encore imparfaites [26]. On reviendra dans la conclusion sur la portée des primitives de contrôle.

3.1. Principe d'exécution

Lorsqu'un module est actualisé (2.2.3.2) et que ses paramètres formels sont liés (2.2.3.4), il peut être exécuté. Les conditions d'exécution sont réunies dans un "index" qui est un objet structuré du langage comprenant une adresse d'unité et un environnement d'exécution. L'environnement d'exécution comprend, en particulier, le locus du module en train d'être évalué, l'état de l'index (actif, en attente ...), les interruptions permises, le chaînage dynamique. Contrairement aux langages de programmation habituels, plusieurs index peuvent être actifs simultanément. Un moniteur joue le rôle de l'unité de contrôle, gère les index et effectue les opérations sur ces index.

3.2. Moniteur

Un moniteur comprend essentiellement une table d'index actifs, avec leur indication de priorité. Cette table est chargée par l'opérateur de contrôle activate (3.3.1.1). L'algorithme du moniteur est donné sommairement ici (l'algorithme formel est donné en 5.10.3).

A3 - (1) Recherche d'un index actif dans la table suivant un certain algorithme que l'on peut redéfinir (algorithme des priorités ou autre)
(2) Vérifier les conditions d'exécution : index non en attente, non bloqué sur un "signal" (3.3.2), etc...
(3) Si les conditions ne sont pas remplies, aller en (1)
(4) Lancer l'évaluation de la partie expression de l'unité repérée par l'index, dans l'environnement de cet index.
(5) Effectuer les opérations de contrôle de l'unité.
(6) Revenir en (1).

Les opérations de contrôle modifient l'état des index de la table, et permettent le déroulement de l'exécution. Les interruptions sont traitées en 3.5.1. Une exécution de module commence par une fonction execute (2.1.1.) qui met un index dans la table du moniteur avec les options standard (3.3.1.1.) excepté pour l'index de retour qui est remplacé par une liste vide, ce qui permet d'arrêter l'exécution du programme.

execute (index, module, suite de paramètres)

3.3. Fonctions de contrôle

On considère trois types de fonctions de contrôle : les primitives, les fonctions de synchronisation, les fonctions composées. Chacune de ces trois catégories est présentée d'une manière informelle, ce qui permettra de mieux comprendre les exemples des parties 3.4 et 3.5. Les définitions formelles sont données en 5.10.2.

3.3.1. Primitives de contrôle

3.3.1.1. Fonction d'activation d'un index

Cette fonction est décrite par :

activate (<locus> . <index> [, <paramètres>])

Le premier argument identifie l'index à mettre dans la table du moniteur - index appartenant à l'actualisation pointée par le locus -.

Les paramètres (facultatifs, comme l'indique l'écriture entre crochets - formalisme de PL/1 -) spécifient les conditions d'exécution et sont identifiés par mots-clés pour permettre les options par défaut.

Liste des paramètres :

(1) ENTRY : <étiq>

<étiq> est une étiquette du module à évaluer qui précise la valeur initiale de l'index (point de début d'exécution)

(2) ETAT : PRET/ATTENTE

Cette option précise l'état de l'index

(3) INTERRUPT : (liste d'interruption)

On reviendra en 3.5.1 pour ce qui concerne les interruptions; les autres options de INTERRUPT sont :

INTERRUPT : ST - Interruptions standard

INTERRUPT : NONE - Aucune interruption n'est permise.

(4) PTY : <constante entière>

La constante entière donne le niveau de priorité pour l'algorithme du moniteur. Dans une implémentation particulière, on donnerait la valeur du niveau maximum, mais ici il n'est pas indispensable de le préciser. La valeur standard est la priorité minimum.

(5) RET : (liste d'index) ou *

L'option RET indique au module appelé quels sont les index à réactiver, ou, plus généralement, à utiliser. Elle explicite le chaînage dynamique tout en le généralisant. La notation * indique qu'il s'agit de l'index courant au module qui s'exécute, ou du seul index du module.

On peut retrouver un index de la liste d'index du RET par la notation :

ret (I, i)

où i est un nombre entier, qui donne le $i^{ème}$ index de l'option RET de l'index I.

ret (I, i, j, k)

donne la liste composée du $i^{ème}$, $j^{ème}$, $k^{ième}$ index ;

ret (*)

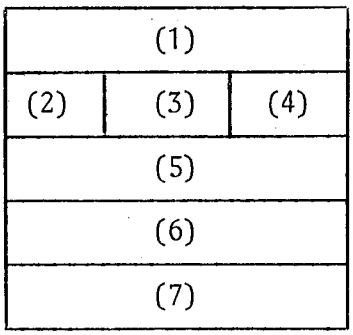
donne la liste de tous les index de l'option RET de l'index courant.

Chaque paramètre a une option par défaut qui permet de simplifier l'écriture dans les cas habituels :

activate (ℓ.I) signifie :
activate (ℓ.I, ENTRY : entry(ℓ), ETAT : PRET, INTERRUPT : ST,
PTY : <valeur standard>, RET : (*))

Comme exemples de fonctions d'activation, on peut avoir :
activate (ℓ.I, ENTRY : E1, ETAT : ATTENTE, RET : (ℓ1.I,ℓ2.J))
activate (ℓ.I, ETAT : ATTENTE, INTERRUPT : NONE, RET : (*,ℓ1.I))

L'index complet est constitué des champs suivants :



(1) étiquette de l'unité à exécuter,
(2) bit prêt ou attente et bit manipulé par les primitives de synchronisation
(3) masque des interruptions
(4) priorité de l'index
(5) locus du module exécuté
(6) nombre d'index de retour
(7) tableau des index de retour.

La primitive activate s'applique à un index n'appartenant pas à la table du moniteur; s'il y est déjà, une erreur est détectée. Par contre, pour les primitives qui suivent, l'index doit être déjà activé, sinon la fonction est sans effet. Donc un index ne peut pas être actif plusieurs fois simultanément.

3.3.1.2. Fonction de mise en attente

stop ([<locus> .] <index> [,[<locus> .] <index>] ...)

La notation syntaxique est toujours celle de la description de PL/1. Cette fonction a pour effet de faire passer l'index ou les index dans l'état ATTENTE. Elle est utilisée pour suspendre l'évaluation d'un module.

3.3.1.3. Fonction de reprise d'exécution

resume ([<locus>]. <index>[,[<locus> .] <index>] ...)

Cette fonction fait passer les index à l'état PRET. Elle permet la reprise de l'évaluation là où elle était arrêtée par un stop.

3.3.1.4. Fonction de désactivation

kill ([<locus> .] <index> [,[<locus> .] <index>] ...)

Cette fonction désactive un index et donc arrête l'exécution du module dirigée par cet index. L'index est enlevé de la table du moniteur et peut donc ensuite être l'objet d'un activate dans un autre contexte d'exécution.

3.3.1.5. Fonctions d'enchaînement d'exécution

Les deux fonctions next et move modifient le pointeur en général appelé compteur ordinal, vers l'unité à exécuter.

next ([<locus> .] <index>)

remplace la valeur du compteur ordinal par la valeur d'étiquette de l'unité suivante; c'est l'exécution séquentielle.

move ([<locus>.] <index> , <étiq>)

charge le compteur ordinal par la valeur d'étiquette; c'est le branchement inconditionnel.

3.3.2. Primitives de synchronisation

Plusieurs index pouvant être actifs simultanément, il s'ensuit que, du point de vue de l'utilisateur, plusieurs modules peuvent être évalués en parallèle. D'où les problèmes de synchronisation. Les évènements pouvant provoquer des attentes dans les modules sont inscrits dans des objets appelés "signaux". Les signaux se définissent par :

def (S, signal)

Les signaux sont schématisés par :

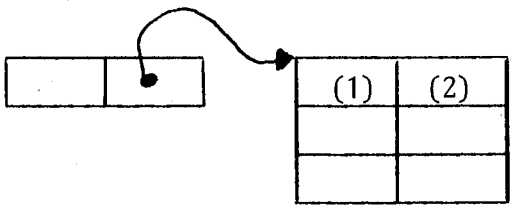


Table de longueur variable

(1) pointeur sur index

(2) compteur.

Ils sont utilisés par les fonctions wait et free qui permettent de bloquer ou de débloquer l'évolution des index sous certaines conditions.

3.3.2.1. Fonction d'attente sur signal

La primitive wait s'écrit sous la forme suivante :

wait (<signal> , <liste d'index> , <findex>)

où <findex> est une fonction quelconque sur les index.

L'algorithme de la fonction wait est :

(A4) - (1) Si l'index courant est dans le signal,
aller à (3) ;
(2) Mettre l'index courant dans le signal avec compteur à 0 ;
(3) Si compteur > 0, il y a interruption d'exécution du module, qui ne pourra reprendre que lorsque son compteur sera $\leq$ 0. La reprise se fera au pas (4).
(4) Si compteur $\leq$ 0, les actions suivantes sont prises : pour chaque index de la liste d'index faire :
(4-1) - Si l'index est dans le signal, aller à (4-3).
(4-2) - Mettre l'index dans le signal avec compteur à zéro
(4-3) - Faire compteur := compteur+1
(5) Exécuter findex .

Exemples :

wait (S, (I,J,K), move (*,ℓ))
wait (S,*, wait (S,, next (*)))

Pour cette dernière instruction, l'index courant est toujours bloqué avant le next, à moins que des fonctions free n'aient déjà été réalisées.

3.3.2.2. Fonction de libération

On écrit : free (<signal>,<liste d'index>)

L'algorithme de free est en partie inverse de celui de wait :

(A5) - Pour chaque index de la liste faire :
(1) Si l'index est dans le signal, aller à (3) ;
(2) Mettre l'index dans le signal avec compteur à zéro ;
(3) Faire compteur := compteur-1.

3.3.3. Fonctions de composition

3.3.3.1. Contrôle conditionnel

Le format de cette instruction est celui de la conditionnelle de Mc Carthy [29].

cond((p$_1$,f$_1$), (p$_2$,f$_2$),...(p$_n$,f$_n$))

où

p$_1$, p$_2$,...,p$_n$ sont des prédicats,
f$_1$, f$_2$,...,f$_n$ sont des fonctions de contrôle.

L'élaboration se fait d'une façon classique :

- si p$_1$ est vrai, f$_1$ est exécuté;
  sinon, si p$_2$ est vrai, f$_2$ est exécuté, etc...

3.3.3.2. Contrôle composé

Il s'écrit : ctl(f$_1$, f$_2$, ..., f$_n$)

où les f$_i$ sont des fonctions de contrôle. Tous les f$_i$ sont exécutés sans tenir compte de l'ordre d'écriture.

Remarque : la composition du ctl et du cond permet de faire une différence entre :

(1) ctl (cond ((p$_1$,f$_1$)) , cond ((p$_2$,f$_2$)) , cond ((p$_3$, f$_3$)))
(2) cond ((p$_1$,f$_1$), (p$_2$,f$_2$), (p$_3$,f$_3$)).

En effet, si au moment de l'évaluation, p$_1$ et p$_3$ sont vrais, alors en (1) on exécutera f$_1$ et f$_3$, par contre, en (2), seul f$_1$ sera exécuté.

3.4. Expressions de contrôle sans parallélisme

Dans les exemples qui suivent, l'équivalent est donné, dans les cas simples, en Algol 68, à moins d'indications contraires. L'astérisque est une dénotation d'index qui signifie l'index courant.

3.4.1. Exécution séquentielle

(ℓ1, expr1, next(*))
(, expr2, next(*))
(, expr3, next(*))
équivalent : ℓ1 : expr1; expr2; expr3;

3.4.2. Exécution collatérale

(ℓ1, (expr1, expr2, ..., exprk), next(*))
équivalent :
ℓ1 : (expr1, expr2, ..., exprk);

3.4.3. Branchements

3.4.3.1. Branchement simple :

(ℓ1,, move(*, ℓ2))
signifie : ℓ1 : goto ℓ2; ou ℓ1 : ℓ2;

3.4.3.2. Branchement conditionnel :

(ℓ1,, cond ((p$_1$, move (*, ℓ2)), (true, move (*, ℓ3))))
signifie : ℓ1 : if p1 then goto ℓ2 else goto ℓ3 fi ;

Remarque : On a un équivalent du test conditionnel de Fortran IV, mais non de Algol 60, car l'action exécutée selon la valeur du prédicat est une fonction de contrôle (branchement) et non une unité (instruction).

3.4.3.3. Aiguillage

```
(ℓ0,, cond((p1, move (*, ℓ1)),
           (p2, move (*, ℓ2)),
            ⋮
           (pn, move (*, ℓn)),
           (true, next(*)))) ( , expr1, ...)
```

signifie : ℓ0 : case p_1, p_2, ..., p_n in
ℓ1, ℓ2, ..., ℓ_n out expr1 esac ;

3.4.4. Groupes itératifs

3.4.4.1. : (ℓ0, expr1,) = ℓ0 : do expr1 ;

```
3.4.4.2. : (ℓ0, store (A, B), next (*))
           (ℓ1,, cond ((sup (val(A), D), move (*,ℓ2)),
                       (true, next (*))))
           (, expr1, next (*))
           (, store (A, plus (val (A), C)), move (*, ℓ1))
           (ℓ2, ...
```

signifie : ℓ0 : for A from B by C to D do expr1 ; ℓ2 : ...

```
3.4.4.3. : (ℓ0,, cond ((p, next (*)), (true, move (*, ℓ1))))
           (, expr1, move( *, ℓ0))
           (ℓ1, ...
```

équivaut à : ℓ0 : while p do expr1; ℓ1 ...

3.4.4.4. : (ℓ0, expr1, cond ((p, next(*))))

L'expression expr1 est répétée tant que le prédicat p est faux. Cette forme d'itération n'a pas d'équivalent en boucle do d'Algol, mais se rapproche du do Fortran où le test sur l'indice est fait après l'exécution du corps.

3.4.5. Appels de sous-programmes

Un module peut jouer le rôle de sous-programme. Soit la définition :

def (SP, mod (,(I), ———))

L'appel à SP se fait avec les étapes suivantes :

(1) actualisation du module identifié par SP,

(2) liaison facultative des paramètres,

(3) arrêt de l'index du module courant, mise en mémoire de l'adresse de retour, activation du module sous-programme SP.

D'où les unités, si SP a un seul paramètre à lier :

(, def (L, locus), next (*))

(, alloc (L, act (SP,)), next (*))

(a)(, bind (L,A), ctl (stop (*), next (*), activate (L.*)))

D'après les options par défaut (cf. § 3.3.1.1.), l'index de SP est dans l'état PRET, positionné à la première unité du module, et l'option RET n'a qu'un index qui est celui du module appelant. Les variantes de l'appel (a) sont :

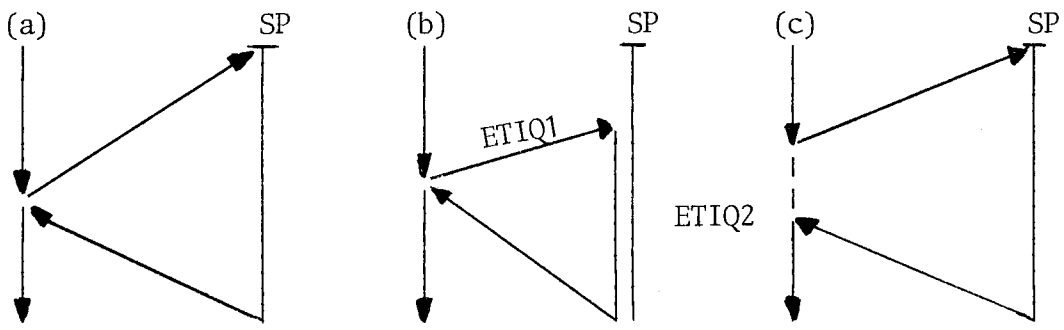
(b) entrée en un point quelconque du module, ce qui généralise les notions d'ENTRY de PL/1 et de FORTRAN IV :

ctl (stop (*), next(*), activate (L.*, ENTRY : ETIQ1))

(c) retour à un point quelconque du programme appelant et non à l'unité suivante :

ctl (stop (*), move (*, ETIQ2), activate (L.*))

En schématisant les flots de contrôle, on a (F8) :



La séquence de retour dans le module sous-programme est la suivante :

—— ctl (kill(*), resume (ret (*)))

Les valeurs de retour d'un sous-programme peuvent se trouver dans les paramètres, ou peuvent être des variables locales de l'actualisation auxquelles on accède par la notation de qualification, avant d'avoir désactualisé.

Exemple :

```
(, alloc (L, act (SP,A)), ctl (stop (*), next (*), activate (L.*)))
(, assign (B, val (L.RES)), next (*)
(, desact (L), ...
```

A et B sont des références locales au module appelant; RES est une variable locale à l'actualisation de L.

3.4.6. Appels récursifs

Nous prenons l'exemple classique de la factorielle dont la définition est le module formel :

```
def (FACT, mod (, (I),
[1]  (, (def (N, fconst), def (L, locus), def (RES, taille (1, 'W'))),
            cond (( eq (N,1), move (*, E1)), (true, move (*, E2))))
[2]  (E1, store (RES, 1),ctl (kill (*), resume (ret (*))))
[3]  (E2, alloc (L, act (FACT, minus (N, 1))),
          ctl (next (*), stop (*), activate (L.*)))
[4]  (, store (RES, mult (N, val (L.RES))), next (*))
[5]  (, desact (L), ctl (kill (*), resume (ret (*))))   ))
```

Explicitation des diverses unites :

[1] Définition d'un paramètre formel (N) et de deux variables locales (utilisation de "taille" (§ 6.1.1.)), puis test sur le cas N=1 ;

[2] Si N=1, on positionne la variable résultat à 1 et retour à l'appelant;

[3] Sinon, actualisation de factorielle avec la valeur N-1 en paramètre et activation du module actualisé.

[4] Utilisation du résultat en calculant :

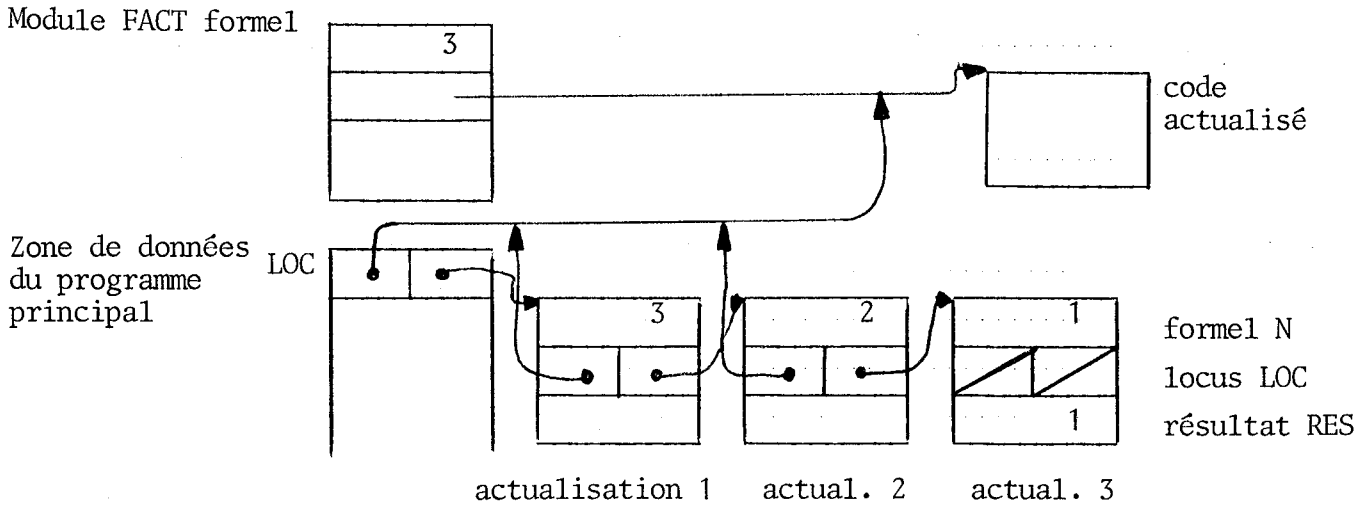
RES := N * résultat (Fact (N-1))

[5] Désactualisation du module factorielle utilisé, et retour à l'appelant.

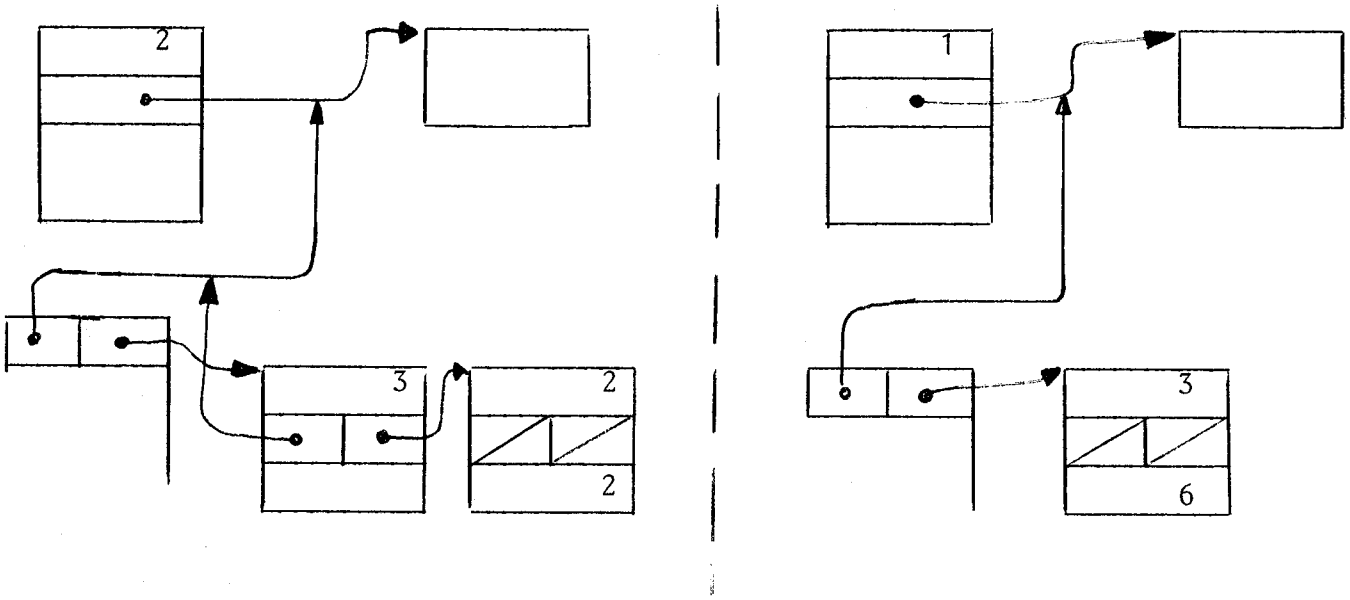
L'appel de factorielle dans un module se fait simplement par :

```
(, def (LOC , locus), ...
(, alloc (LOC, act (FACT, 3)),
          ctl (next (*), stop (*), activate (LOC.*)))
   :
utilisation du résultat par LOC . RES
   :
(, desact (LOC), ...
```

Dans le cas de l'appel de factorielle (3), les actualisations successives donnent, au niveau le plus profond : (F9) :



On a ensuite les états :



3.4.7. Coroutines

Les coroutines ont été introduites par Conway [11] et développées par Mc Ilroy [31]. Ce sont des modules qui s'appellent sans hiérarchie et peuvent s'exécuter "par morceaux" en reprenant l'exécution là où on les avait laissés.

On donne ici un exemple simple où une coroutine comptabilise les appels entre un programme principal et un sous-programme. Un exemple plus complexe, utilisant l'actualisation et la récursivité est donné au § 6.4.

Définition du module principal :

```
def (M1, mod (,(I),
    (, (def (LOC1, locus), def (LOC2, locus),
       def (COMPTEUR, taille (1, 'W'))), next (*))
    (, (alloc (LOC1, act (TRANSIT, COMPTEUR)),
       alloc (LOC2, act (M2))),
            ctl (activate (LOC1.*, RET : (*, LOC2.*)),
                activate (LOC2.*, ETAT : ATTENTE), stop (*), next (*)))
    (L1, bind (LOC2, paramètres), ctl (stop (*), next (*), resume (LOC1.*)))
    (, traitement 1, cond ((p1, move (*,L1)),
              (true, ctl (next (*), kill (LOC1.*, LOC2.*)))))
    (, (desact (LOC1), desact (LOC2)), kill (*))  ))
```

La première unité donne les définitions des variables locales; la deuxième initialise par actualisation les modules de transit et M2, les active et lance l'initialisation du module transit.

Puis vient le traitement proprement dit à partir de ℓ1, par la liaison des paramètres de M2 actualisé et son appel via transit; au retour des sous-programmes, on effectue un traitement, et suivant le test sur p_1, il y a un retour à ℓ1 ou un arrêt du programme avec désactualisation des modules appelés.

Module de transit :

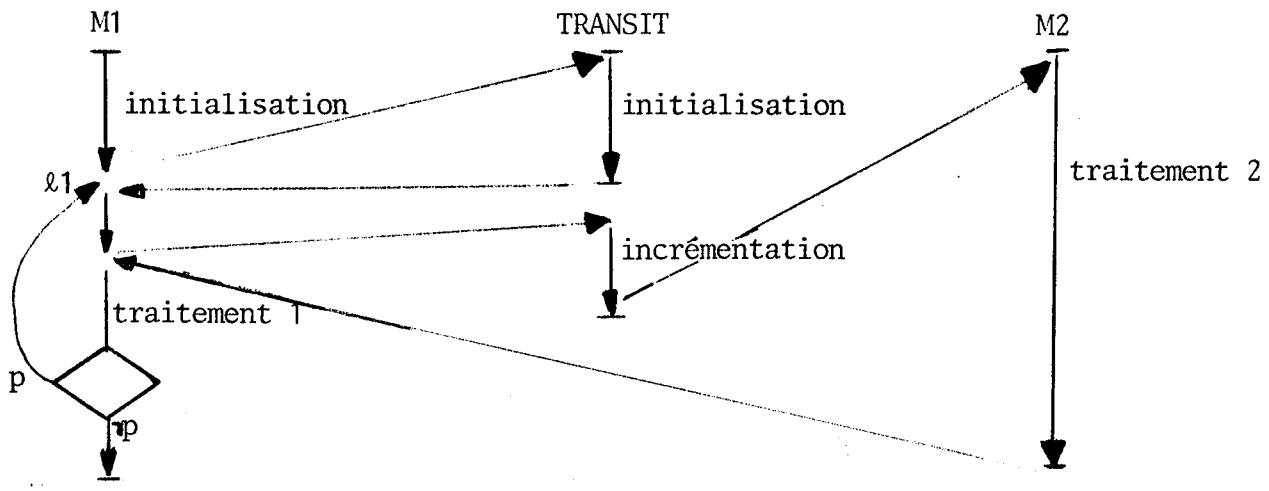
```
def (TRANSIT, mod (, (I),
    (, def (R, fref), next (*))
    (, store (R, 0), ctl (stop (*), next (*), resume (ret (*, 1))))
    (, store (R, plus (val (R), 1)), ctl (stop (*), resume (ret (*, 2))))))
```

Les deux premières unités font l'initialisation tandis que la troisième est celle sur laquelle se fait l'incrémentation.

Le module M2 n'a rien de bien particulier :

```
def (M2, mod (, (I),
     (E1, ...
     (, traitement 2, ctl (stop (*), move (*, E1), resume (ret (*)))) ))
```

Schéma du flot de contrôle : (F10) :



3.4.8. Structure de bloc

Depuis ALGOL 60, la plupart des langages de programmation de haut niveau utilisent la structure de bloc à cause de nombreux avantages que nous ne détaillons pas ici. Un bloc peut être simulé par un module interne que l'on actualise immédiatement.

Soit par exemple :

```
def (PROG, mod (, (I),
     :
[1]  (, def (B1, locus), next (*))
```

```
[2]  (, alloc (B1, act (mod (, (J),
                (, def (N, ref (...)), next (*))
                   :                                  } Bloc B1
                (, ——— , ctl (kill (J), resume (ret (*))))
             ))), ctl (stop (*), next (*), activate (B1.*)))

[3]    (, desact (B1), next (*))
```

Commentaires :

[1] Définition du locus appartenant au programme de niveau 0 qui pointera sur l'actualisation du bloc.

[2] Actualisation du module non identifié qui simule le corps du bloc B1. La dernière unité du module désactive l'index et rend le contrôle au niveau 0. N est défini comme une variable locale à ce bloc. La dernière ligne est l'action de contrôle de l'unité qui a fait l'actualisation; elle consiste à arrêter l'index de niveau 0 pour aller exécuter l'actualisation qui vient d'être faite, c'est-à-dire le bloc B1.

[3] A ce point, le bloc a été exécuté, le locus peut être libéré, ainsi que la place prise par l'actualisation, en faisant la désactualisation.

Cette forme d'écriture des blocs donne un bel exemple d'extension par macros (§ 6.3.).

L'implémentation d'un module interne est faite de telle sorte que la zone de données correspondante comporte une mémoire contenant le locus du module englobant. Comme on sait, lors de la recherche des identificateurs à quel niveau se trouve une définition, il suffit de remonter dans la chaîne statique des locus pour avoir la valeur de l'identificateur. On traite dans le chaître 6 d'un dialecte avec structure de blocs, des chaînages statiques et dynamiques à l'appel de modules (sous-programmes) récursifs, ainsi que du problème du goto à l'extérieur des blocs.

3.4.9. Réentrance

D'après le principe de l'actualisation qui a été exposé au § 2.2.3.2, il n'y a pas de problème pour la réentrance; il suffit, pour chaque utilisation d'un module partagé de faire une actualisation, avec paramètres modified, s'il y a des paramètres.

3.4.10. Marche arrière

Cette technique de la "marche arrière" (backtraking) a été étudiée [16] [21] pour résoudre les problèmes où plusieurs possibilités peuvent être tentées avant de trouver la solution comme c'est le cas par exemple pour l'analyse syntaxique non déterministe. Parmi les nombreuses méthodes d'écriture qui peuvent être envisagées, il y a une façon simple d'arriver à ce résultat en employant deux index dans le module et en utilisant des macros syntaxiques qui permettent de générer les unités intermédiaires à condition de traiter convenablement le programme au préalable. Un exemple complet est donné au § 6.5.

Le principe de la "marche arrière" est le suivant : à partir d'un certain point, il y a un choix entre plusieurs solutions. Les alternatives de ce choix sont prises par une fonction choix et chacune est essayée dans la suite de l'algorithme. Lorsqu'une solution s'avère mauvaise, on tombe dans l'état d'échec, et l'algorithme est repris en sens inverse pour restaurer les valeurs initiales jusqu'au point du choix où une autre solution est tentée. Lorsque la solution qui est supposée unique est trouvée, on passe à une autre partie de l'algorithme.

3.4.11. Branches de programme indexées

L'exemple de la "marche arrière" a fourni l'occasion d'utiliser plusieurs index dans un même module. Une autre application immédiate est de regrouper des parties de programme identiques et éclater suivant les différents cas sans avoir à faire de tests, ce qui, a priori, doit augmenter l'efficacité d'un même programme écrit avec un seul index.

Dans l'exemple qui suit, infeg est le prédicat $\leq$.

```
def (SOMME, mod (,(SIMPLE, CARRE, CUBE),
[1]  (ℓ0, (def (X, taille(1, 'W')), def (R, taille (1, 'W')),
          def (S, taille (1, 'W'))), next (*))
[2]  (ℓ1, (store (X, 0), store (R, 0), store (S, 0)), next (*))
[3]  (ℓ2, store (X, plus (val (X), 1)), next (*))
[4]  (ℓ3, store (R, val (X)), ctl (move (SIMPLE, ℓ6), move (CARRE, ℓ5),
                                   move (CUBE, ℓ4)))
[5]  (ℓ4, store (R, mult (val (R), val (X))), next (CUBE))
[6]  (ℓ5, store (R, mult (val (R), val (X))), ctl (next (CARRE), next (CUBE))
[7]  (ℓ6, store (S, plus (val (S), val (R))),
          cond ((infeg (val (X), fconst), move (*, ℓ2)),
                (true, ctl (kill (*), resume (ret (*))))))))
```

Commentaires :

[1] Définitions des variables locales; [2] Initialisations ;
[3] Incrémentation de X ; [4] Initialisation du terme de la somme;
[5] [6] produits sur R (schéma de Horner); [7] calcul de la somme partielle et test sur la fin de la somme, avec une variable formelle représentant l'indice du terme maximum. Le terme * signifie l'index courant, donc celui des trois qui est actif (cf. § 5.10.3).

Les séquences d'appel donnent :

(a) (, alloc (LOC, act (SOMME, val (N))),
ctl (stop (*), next (*), activate (LOC . SIMPLE)))
le résultat dans LOC.S est : $\sum_{X=0}^{N} X$;

(b) Avec le contrôle : ctl (stop (*), next (*), activate (LOC.CARRE))
on a : LOC.S = $\sum_{X=0}^{N} X^2$;

(c) Avec le contrôle : ctl (stop (*), next (*), activate (LOC.CUBE))
on a : LOC.S = $\sum_{X=0}^{N} X^3$.

Les possibilités d'écriture de programmes avec plusieurs index restent encore à exploiter.

Remarques : On peut se demander comment utiliser les primitives de contrôle pour résoudre un problème particulier. Quelques unes sont faciles à manipuler (next, move) ; d'autres peuvent poser des questions, en ce sens que, étant très élémentaires, elles n'ont qu'une sémantique réduite et leur composition en fonctions plus complexes n'est pas évidente. Les principaux cas ont été présentés :

- appel de sous-programme : ctl (stop (*), next (*), activate (——))
- retour de sous-programme : ctl (kill (*), resume (ret (*)))
- appel et retour de coroutine : ctl (stop (*), next (*), resume (——))
- conditionnelle (if then else) : cond ((p_1, f_1), (true, f_2))

On peut se contenter de s'en tenir à ces formes, mais les possibilités de combinaison restent innombrables, notamment si l'on inclut les primitives d'interruption (3.5). Il reste à démonter alors que le programme est "correct", c'est-à-dire "calcule" exactement ce que l'on souhaite. Ce souci devient d'autant plus pressant que, l'écriture du programme n'étant pas transparente, on n'a pas la certitude d'atteindre son but. La façon de procéder est, à mon avis, la suivante : dans un premier temps, étude du graphe de contrôle pour éliminer les erreurs sémantiques (in-

compatibilités entre certaines primitives, branches de programme non accessibles, ...). Cette étude est relativement simple à entreprendre et peut se traduire en conditions sur les primitives elles-mêmes ou sur le graphe de contrôle issu de ces primitives [5]. Dans un deuxième temps on peut modifier le graphe de contrôle par des transformations simples et essayer de déduire des propriétés sur les programmes eux-mêmes ce qui donnerait des "preuves de programmes", mais il semble que ce domaine en soit encore à des balbutiements.

3.5. Problèmes de parallèlisme

3.5.1. Interruptions

On considère deux types d'interruptions : les interruptions synchrones et les interruptions asynchrones. Les premières sont essentiellement tous les types d'interruptions survenant de conditions particulières dans l'exécution des programmes : division par zéro, débordement dans les opérations arithmétiques, etc. Ce sont celles qui sont traitées par l'option INTERRUPT du activate (cf. § 3.3.1.1). Ses interruptions sont répertoriées et connues de l'utilisateur par mots-clés. Pour chacune, un traitement standard est écrit et utilisable directement. C'est ce traitement qui est spécifié par l'option INTERRUPT :ST. Si l'on ne veut pas traiter les interruptions de ce type, on écrit INTERRUPT : NONE. Lorsqu'une condition exceptionnelle intervient dans l'exécution d'une unité, il y a simplement arrêt de l'exécution du programme et désactualisation de tous les modules. Si, par contre, on veut traiter les interruptions d'une manière personnelle, on peut redéfinir les modules de traitement pour chaque interruption répertoriée. Cela se fait en définissant un module d'interruption, au même titre qu'un module quelconque.

Exemple : def (NEWZERODIVIDE, mod (————))

On utilise ce module dans l'option INTERRUPT de la manière suivante :

INTERRUPT : (zerodivide (NEWZERODIVIDE, <index> , <liste de paramètres>),
overflow (...))

L'effet sémantique est : zerodivide est l'identificateur - réservé dans ce contexte - d'un locus qui contient par défaut une actualisation du module standard. La redéfinition dans INTERRUPT signifie :

alloc (zerodivide, act (NEWZERODIVIDE, <liste de paramètres>))

Lorsqu'une condition de division par zéro survient dans le programme, le moniteur vérifie que cette condition soit en fonction pour l'index courant et exécute alors :

ctl (stop (*), activate (zerodivide . <index> , INTERRUPT : NONE))

On note que l'index courant est arrêté sur l'unité qui a provoqué l'interruption, et que le module d'interruption ne peut être interrompu.

Le deuxième type d'interruption concerne en particulier les entrées-sorties. Ces interruptions proviennent du fait que plusieurs processus peuvent se dérouler simultanément et qu'une coordination est nécessaire. Elles rentrent dans le cadre général des processus parallèles et peuvent être décrites par wait et free (§ 3.5.2.4). Une étude un peu semblable est faite dans [50]. Les opérations d'entrées-sorties proprement dites sont du ressort de l'interpréteur (Annexe 1.4).

3.5.2. Synchronisation de processus

Un module peut activer plusieurs index de modules différents tout en restant d'ailleurs lui-même actif, ce qui a pour effet d'exécuter plusieurs modules en parallèle ou d'une manière indépendante les uns des

autres. Dans ces cas, il est important de pouvoir synchroniser certaines parties de modules ou de suspendre l'exécution de l'un d'entre eux jusqu'à ce que certaines conditions soient remplies. C'est dans ce but que l'on utilise les fonctions wait et free ainsi qu'il sera montré dans les exemples qui suivent.

3.5.2.1. Section critique non partageable

On appelle ici section critique non partageable une partie de programme qui doit être exécutée entièrement sans que d'autres programmes puissent s'exécuter pendant ce temps. C'est le cas par exemple d'une consultation de table qui n'existe qu'à un seul exemplaire en mémoire. Pendant tout le temps de la consultation, les autres programmes ne doivent pas modifier la table, ce qui provoquerait des résultats aberrants. Dans l'exemple ci-dessous, on pourra supposer que SC1 est la consultation de table, et SC2 la modification.

L'appel en parallèle des deux modules se fait par :

```
(, (def (LOC1, locus), def (LOC2, locus)), next (*))
(, (alloc (LOC1, act (M1, LOC2)), alloc (LOC2, act (M2, LOC1))),
   ctl (activate (LOC1.*), activate (LOC2.*), stop (*), next (*)))
```

Dans une définition englobante on trouve :

```
def (S, signal)
```

Les modules sont définis par :

```
def (M1, mod (, (I), ...
        (, def (FL1, floc), ...
         :
         :
        (ℓ1, ——, wait (S, (FL1.*), next (*)))
                    |
                    | SC1
        (ℓ2, ——, ctl (free (S, (FL1.*)), next (*))) ...
```

```
def (M2, mod (, (J), ——
      (, def (FL2, floc), ...
          ⋮
      (ℓ'1, ——, wait (S, (FL2.*), next (*)))
                  | SC2
      (ℓ'2, ——, ctl (free (S, (FL2.*)), next (*))) ...
```

3.5.2.2. Opérations P et V de Dijkstra [13]

Les opérations P et V sur les sémaphores sont bien connues et largement utilisées dans les systèmes d'exploitation à multitraitement. On peut les simuler facilement comme le montre l'exemple du producteur et du consommateur. Le producteur charge un stock et le consommateur utilise les objets du stock en se mettant en attente lorsque le stock est vide (on peut, de la même manière, arrêter le producteur lorsque le stock est plein).

```
def (P, mod (, (I), ——
      (ℓ0, def (CONSOMMATEUR, floc), ——
      (ℓ1,
            | Production d'un objet 0 et mise en stock
      (ℓn, ——, ctl (free (S,(CONSOMMATEUR.*)), move (*, ℓ1))) ))
def (C, mod(, (J), ——
      (ℓ'0, def (CONSOMMATEUR, floc), next (*))
      (ℓ'1,, wait (S,(CONSOMMATEUR.*), wait (S, (), next (*))))
            |
            | Utilisation d'un objet 0 du stock formé par producteur
      (ℓ'n, ——, move (*,  ℓ'1)) ))
```

L'appel des modules est fait par :

```
      (, (def (LOC1, locus), def (LOC2, locus)), next (*))
      (, (alloc (LOC1, act (P, LOC2)), alloc (LOC2, act (C, LOC2))),
         ctl (stop (*), next (*), activate (LOC1.*), activate (LOC2.*)))...
```

3.5.2.3. Synchronisation

Lorsqu'un module lance deux processus en parallèle, il faut généralement attendre que les deux soient terminés pour que le module appelant reprenne le contrôle, s'il ne peut lui-même être partagé. D'où le problème de la synchronisation qui est quelquefois traité différemment dans les systèmes par l'intermédiaire d'un "scheduler" et d'un mécanisme de mise en attente des tâches.

Module appelant :

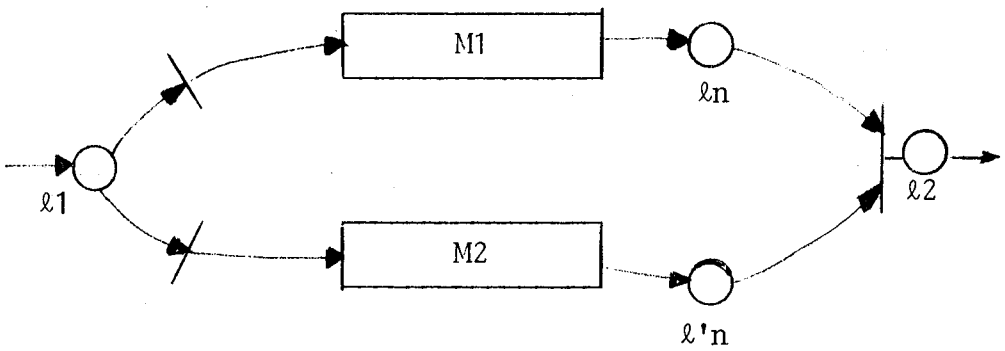
```
mod (, (I), (ℓ0, (def (LOC1, locus), def (LOC2, locus)), next (*))
            (ℓ1, (alloc (LOC1, act (M1, LOC2)),
                  alloc (LOC2, act (M2, LOC1))),
            ctl (activate (LOC1.*), wait (S, (LOC1.*),),
                 activate (LOC2.*), wait (S, (LOC2.*),),
                 stop (*), next (*)))
            (ℓ2,...
```

Synchronisation des modules appelés

```
 def (M1, mod (, (I), (, def (FL, floc), ——
                  |  corps du module
          (ℓn,, ctl (wait (S, ()), kill(*)), free (S, (FL.*)))) ))
 def (M2, mod (,(J), (, def (FL, floc), ——
                  |  corps du module
          (ℓ'n,, ctl (wait (S, (), ctl (kill (*), resume (ret(*)))),
                      free (S, (FL.*)))) ))
```

N'importe lequel des deux modules pourrait faire le resume sur l'index du module appelant, car les deux en sont au même point une fois que le wait final est débloqué.

Cette synchronisation fait penser aux "transitions" des "filets de Petri" (Petri nets [33]). En effet, on peut grossièrement représenter les modules de l'exemple précédent sous forme de "filets de Petri" où les boîtes des modules représentent l'ensemble des actions accomplies dans les modules, et où peuvent se déclancher des retards. (F11) :



3.5.2.4. Exécution asynchrone

Il arrive souvent dans les systèmes d'exploitation qu'un module lance une action qui se déroule en même temps que le module lui-même, au moins pendant un certain temps. C'est le cas en particulier des entrées-sorties où une action de lecture physique est lancée avant que le programme n'ait effectivement besoin de la donnée contenue dans l'enregistrement lu. Dans l'exemple qui suit, on a illustré l'appel d'un module de lecture et l'attente éventuelle du programme principal s'il arrive au traitement de l'enregistrement lu avant que la lecture ne soit terminée.

Module principal :

```
mod (, (I), ——
  (, def (LOC, locus), ——
  (ℓ1, alloc (LOC, act (LECT, ...)),
        ctl (activate (LOC.*), next (*), wait (S, (I),)))
        |suite du programme
  (ℓ2, —— , wait (S, (), next (*)))
        |traitement de l'enregistrement lu.
```

Module de lecture :

```
def (LECT, mod (, (J), ——
        |lecture enregistrement
    (ℓ'n,, ctl (kill (*), free (S, (ret (*))))) ))
```

CHAPITRE IV

4. CONTROLE DES TYPES A LA COMPILATION - MECANISME DES CLASSES

4.1. Les types ("data types") dans les langages de programmation

Les types sont introduits dans les langages de haut niveau e
les valeurs de ces types sont les objets du langage. Exemple, Algol 60 re
connaît le type integer et l'on peut déclarer des objets integer :

integer X, Y ;

ou avoir une valeur de type integer :

X + Y

ou déclarer une procédure qui retourne une valeur integer.

Le rôle joué par les types est multiple [24]:

(1) Spécifier la représentation externe des objets ;
(2) Spécifier la représentation interne et donner l'information pour le codage des opérations ;
(3) Définir les règles de manipulation des objets (conversions) ;

En ALGOL 60, toutes ces fonctions sont groupées; dans un langa
comme PL/1, les attributs, qui remplacent les types essaient de définir
précisément la représentation interne.

Exemple : Dcl X fixed dec ;
Dcl Y fixed bin ;

X et Y ne diffèrent que pour les points 2 et 3.

Dcl A fixed bin (10) ;
Dcl B fixed bin (12) ;

A et B ne diffèrent que dans l'utilisation (3) et non dans la représenta

(A, B = 4 ; donnera la même représentation en mémoire sur un demi-mot).

La représentation interne peut être étudiée d'une façon plus formelle que par un langage de base [43] [22] [3] mais ce n'est pas notre propos d'en parler ici.

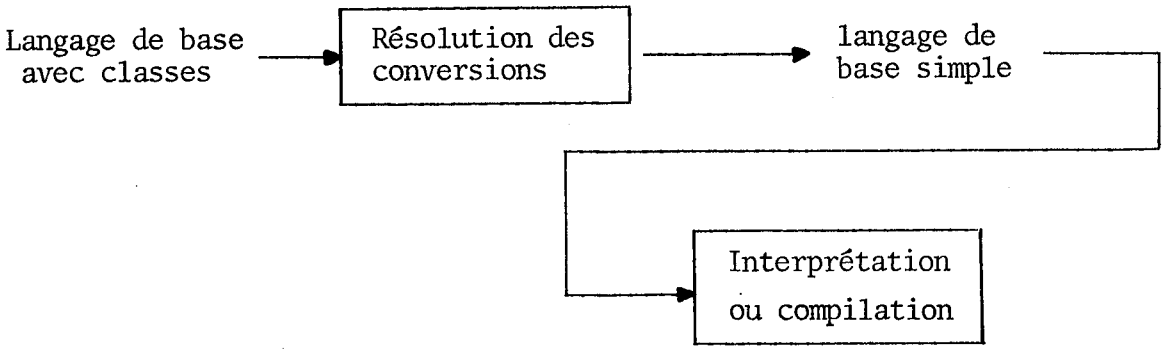
En Algol 68, les modes jouent aussi le même rôle, mais on peut dire que les modes de base : int, bool, char ... définissent la représentation interne (et externe par les dénotations) et le codage des opérations, tandis que les constructeurs (struct, ref, []) définissent une structure de données, donc indiquent les règles de manipulation et l'accessibilité. Les autres constructeurs : union et proc ont des significations différentes.

Dans ce chapitre, on essaie de formaliser et de généraliser le rôle des types en séparant les notions d'utilisation de celles de la représentation. Les objets du langage de base sont décrits en termes de représentation interne. Donc si l'on veut spécifier qu'une famille d'objets de base a des propriétés particulières, par exemple, être les opérandes d'une certaine "procédure" ou pouvoir être transformés par une "conversion" en une autre famille d'objets de base, il faut définir des relations entre des groupes d'objets de base. Pour cela, on définit abstraitement des ensembles d'objets appelés classes et des relations qui sont :

- pour un objet : relation d'appartenance à une classe C
- pour deux classes : . la relation d'inclusion avec les différents cas du § 4.2.2.;
  . la relation de conversion ;
  . la relation de procédure.

Ces relations donnent une structure non hiérarchique de transformations [25] à l'inverse de Algol 68 et conduisent à l'étude d'un graphe de conversions entre les classes (cf. § 4.2.3 et suite). Ce graphe est utilisé pour vérifier les types et générer les conversions au moment de la compilation,

donc dans un premier passage. Ceci est important car on pourrait concevoir des conversions ou des modes définis dynamiquement, ou s'appliquant à un certain niveau de passage comme IMP [23]. Ici on suppose un traitement en deux passages (cf. § 2.1.1.).



Remarques : L'idée de traiter les conversions et tout ce qui s'y rattache par les classes est de Ph. JORRAND ; la mise au point du mécanisme général a été faite en commun dans le cadre du projet dont on a parlé dans l'introduction. Je me suis occupé plus particulièrement de la partie algébrique de définition des opérations et de la formulation algorithmique.

4.2. Les classes

4.2.1. Création d'une classe

Pour un module, on peut donner de façon statique un ensemble de définitions qui conduisent à la création d'un certain environnement de contrôle à la compilation. Ces définitions sont essentiellement les classe et les relations entre ces classes. On définit une classe par :

def (identificateur de classe, class)

Exemple : def (entier, class)
def (station de ski, class)

Les classes introduites de cette manière sont indépendantes les unes des autres.

4.2.2. Opérations d'ensemble sur les classes

Les classes étant des ensembles d'objets, on peut en créer de nouvelles par des opérations d'ensemble ; toutes ces opérations conduisent à mettre en évidence la relation d'inclusion ou d'exclusion entre les classes.

4.2.2.1. Inclusion

Une classe peut être définie explicitement comme incluse dans une autre déjà existante. Si C1 est une classe,

def (C2, in C1)

donne une nouvelle classe C2 avec la relation C2 $\subseteq$ C1, c'est-à-dire que les éléments de C2 ont les propriétés de C1 (mais peuvent en avoir en propre). Nous noterons la relation d'inclusion par :

i (C2, C1)

Exemple : def (pair, in entier)

def (arbre, in forêt)

où forêt est une classe qui sera définie en 4.2.5.

4.2.2.2. Union

On peut définir une nouvelle classe comme étant l'union d'une liste de classes :

def (C, union (C1, C2, ..., Cn))

Dans ce cas, tous les C_i ont les propriétés que l'on peut donner à C ; cela se traduit par les relations :

i(C1, C) , i (C2, C) ...

Exemple : def (montagne, union (station de ski, forêt))

4.2.2.3. Intersection

Si une classe doit avoir les propriétés de plusieurs autres, elle peut être considérée comme l'intersection de la liste de ces classes

def (C, inter (C1, C2, ..., C_n))

C'est une nouvelle classe qui est en relation d'inclusion avec les classes C_i :

i (C, C1) , i (C, C2) ...

Exemple : def (scierie, class)

def (réserve de bois, inter (forêt, scierie))

4.2.2.4. Complément

Le complément est un opérateur qui permet de spécifier l'exclusion de deux classes. Ainsi :

def (C, compl (C1, C2))

indique que C est une nouvelle classe complément de C1 par rapport à C2, c'est-à-dire qu'elle est incluse dans C2, mais ne peut avoir aucune des propriétés de C1. On note la relation d'exclusion par $\tilde{i}$, d'où les relations :

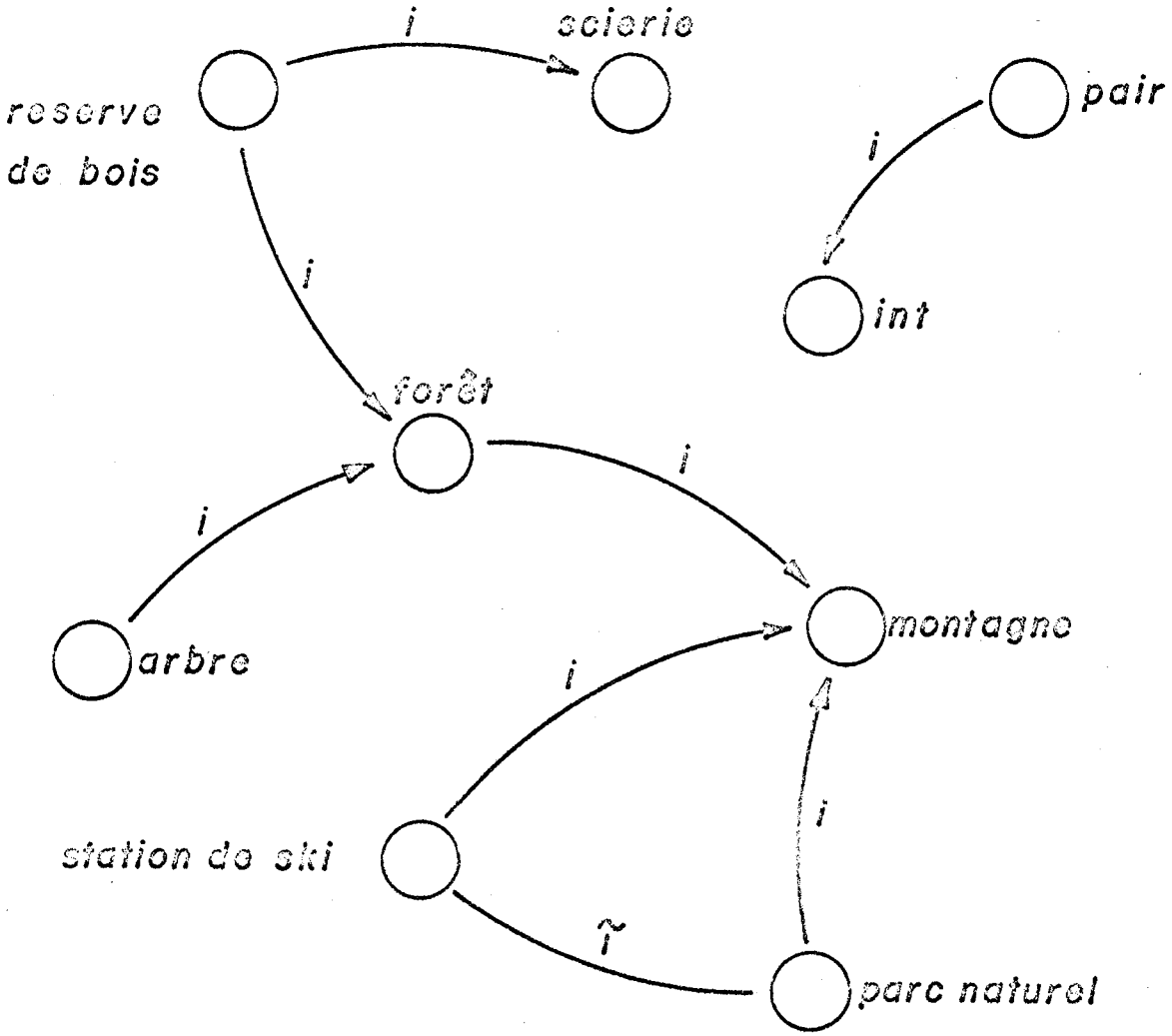
$\tilde{i}$ (C, C1) qui est symétrique

i (C, C2)

Exemple : def (parc naturel, compl (station de ski, montagne))

4.2.3. Graphe des classes

Les relations i et $\tilde{i}$ permettent de construire un graphe dans lequel les classes sont les noeuds et les relations sont des arcs orientés (i) ou non orientés ($\tilde{i}$). Si l'on reprend les exemples donnés, on a le graphe (F12) :



Sur le graphe, les relations sont décomposées et les opérateurs qui les ont introduites (union, inter ...) n'apparaissent pas ; ainsi on a simplement, par exemple :

i (forêt, montagne)

i (station de ski, montagne)

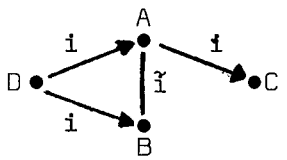
i (parc naturel, montagne)

4.2.4. Classes vides

L'existence de ĩ-arcs amène le problème de décider si les classes construites sont cohérentes entre elles. Dans l'exemple suivant où B, C sont des classes :

def (A, compl (B, C))

def (D, inter (A, B))



il est évident que D n'est pas cohérent, car d'une part, il a les propriétés de A, d'autre part celles de B, mais A et B ne peuvent avoir de propriétés communes, donc D ne peut contenir d'éléments.

Plus généralement, on détermine les classes "vides" par la proposition :

soit I la fermeture transitive de la relation i
et K(C) = {C' tel que I(C, C')}
alors
[($\exists$ C1, C2 tels que C1, C2 $\in$ K(C)) $\wedge$ $\tilde{I}$(C1, C2)]
$\Longleftrightarrow$ C est vide.

4.2.5. Produit cartésien

Etant donnée une liste de classes, on peut définir une nouvelle classe qui en est le produit cartésien :

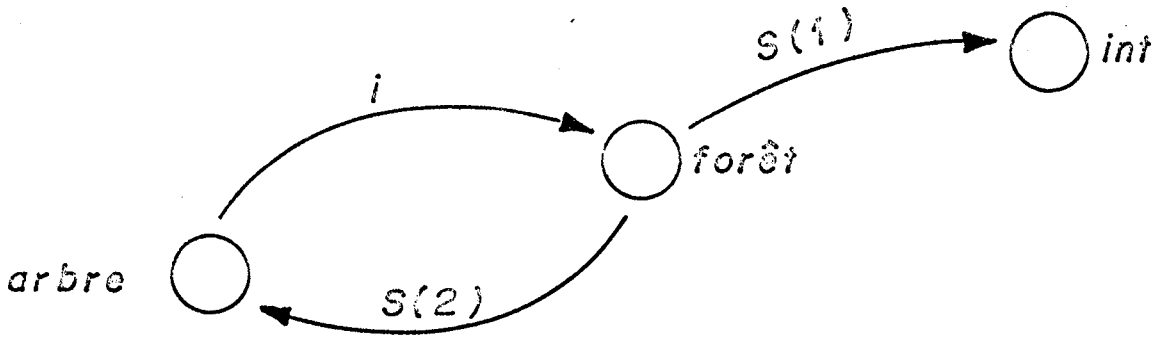
def (C, cart (C1, C2, ..., Cn))

Les éléments de C sont une liste d'objets dont chacun appartient à la classe correspondant à son indice. Sur le graphe des classes, le produit cartésien introduit un nouveau type d'arcs qui sont les sélecteurs et notés S (indice) - arc.

Ainsi on peut définir forêt avec deux composantes : une donnant la superficie en hectares, une autre donnant l'espèce principale d'arbres :

def (forêt, cart (entier, arbre))

ce qui complète le graphe :



4.3. Relations fonctionnelles

Les relations issues des opérations d'ensemble n'impliquent pas de transformations sur les objets. Par contre, les relations fonctionnelles établissent un processus de passage d'une classe à une autre. Ces relations sont les procédures et les conversions. La différence entre ces deux types de transformations est que les procédures doivent être explicitement appelées dans le programme tandis que les conversions sont appliquées automatiquement, sous certaines conditions de "niveau", lorsque l'évaluation le demande.

4.3.1. Les conversions

Les conversions sont définies par le quadruplet :

conv (l, C1, C2, L)

où l est le niveau de la conversion (cf. 4.3.2.),

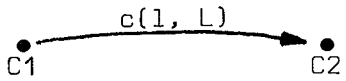
C1 la classe de l'objet argument,

C2 la classe de l'objet résultat,

L le processus de transformation.

Puisque le langage de base a été décrit au chapitre 2, L est en fait un opérateur de l'interpréteur à un paramètre — éventuellement une liste si la classe C1 est un cart — et qui donne un résultat.

Pour le graphe des classes, une conversion est un arc orienté entre C1 et C2 qu'on notera "c(l, L) - arc" :



Une conversion apparaît dans une définition :

def (déboisement, conv (l1, forêt, réserve de bois, op 1))

Cependant, comme une conversion n'est pas , en général, appelée explicitement, le nom peut être omis :

def (, conv (l2, parc naturel, station de ski, op 2))

4.3.2. Les niveaux

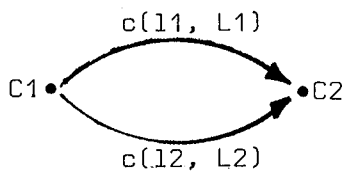
Les conversions sont classées par niveau ; les niveaux forment un ordre partiel qui permet de préciser quelle conversion il y a lieu d'appliquer lorsque plusieurs sont possibles. Une définition de niveau se fait par :

def (l1, level)

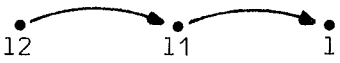
L'ordre partiel est introduit par in de la même manière que pour les classes :

def (l2, in l1)

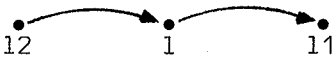
Parallèlement au produit cartésien des classes, on peut avoir un produit cartésien de niveaux. Le principe du choix des conversions est le suivant : soient les niveaux l1 et l2 précédemment définis, et deux classes C1 et C2 reliées par deux conversions :



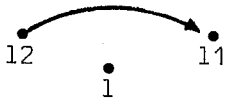
On suppose que l'on veut un objet de classe C2 comme opérande d'une procédure p et que la classe a priori de cet opérande est C1. Il y a donc à utiliser une conversion. La procédure p impose un niveau de conversion 1 (cf. 4.3.3.). Alors plusieurs cas peuvent se présenter ; soit le graphe des niveaux est :

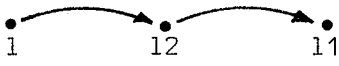


alors la conversion c(11, L1) est prise ;

soit il est : 

et on effectue la conversion c(12, L2) ; soit il se présente comme :



ou 

et il n'y a pas de conversion possible.

On remarque que l'on généralise ainsi les coercions d'Algol 68 où les "niveaux" sont les contextes (positions syntaxiques) fort, ferme, faible, mou, et où l'on a par exemple l'opération d'affectation (:=) qui impose un contexte fort à droite et mou à gauche, ou bien les opérandes d'expressions arithmétiques qui sont dans un contexte ferme. Les opérations algébriques précises des conversions et des compositions de conversion et d'inclusion sont données en 4.5.3.

4.3.3. Les procédures

Les procédures se présentent comme des conversions, c'est-à-dire par le quadruplet :

proc (k, C1, C2, L)

où k est le niveau de conversion imposé sur l'argument,

C1 la classe de l'argument,

C2 la classe du résultat,

L l'opération à effectuer, qui est aussi un opérateur.

On peut définir par exemple :

def (protection de la nature,

proc (l1, montagne, parc naturel, op 3))

def (chiffre d'affaires,

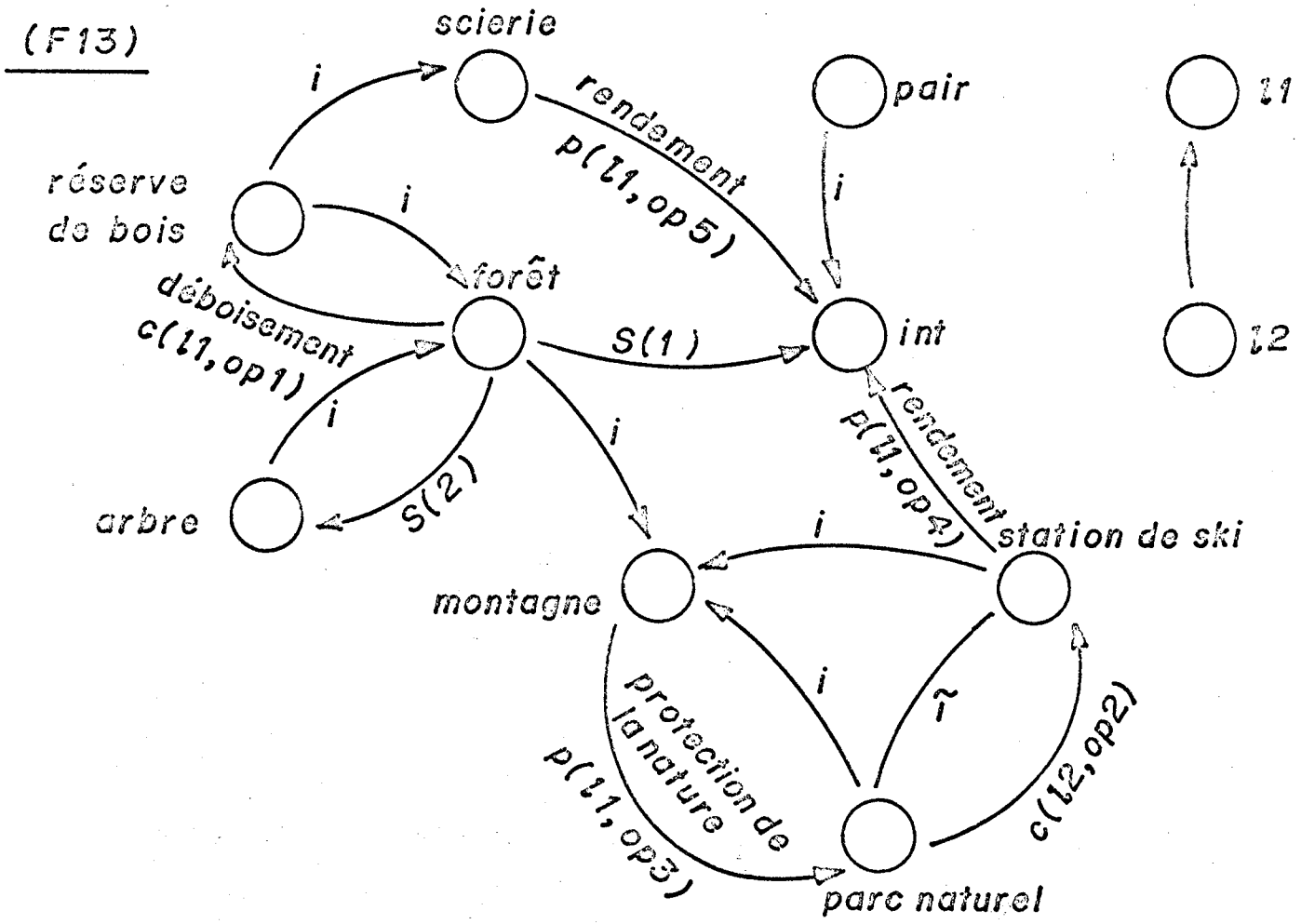
proc (l1, station de ski, entier,op 4))

Plusieurs procédures peuvent porter le même nom à condition qu'elles puissent être distinguées par les classes des arguments (procédures génériques) :

def (chiffre d'affaires, proc (l1, scierie, entier, op 5))

Une procédure est notée comme un p(k, L)-arc entre les classes C1 et C2.

Le graphe complet se compose de (F13) :



4.4. Appartenance à une classe

Les objets de base sont les éléments des classes. Lorsqu'on veut spécifier cette appartenance, on écrit :

as (C, S)

où C est une classe, et S un objet de base, et qui signifie que S pourra être utilisé comme appartenant à la classe C. Les opérations — procédures ou conversions — partant de la classe C seront donc applicables à l'objet "classifié" ainsi créé.

Exemples : def (Vanoise, as (parc naturel, objet parc naturel))

def (Fontainebleau, as (forêt, objet forêt))

def (I, as (entier, taille (1, 'W'))) (cf.2.2.1.).

ce dernier exemple définit I comme une référence ayant les propriétés des entiers.

4.5. Utilisation des graphes

4.5.1. Principe du contrôle des classes

Lors de l'appel des procédures, un contrôle doit être fait à la compilation pour vérifier si la classe du paramètre effectif correspond à celle de l'argument. L'opérateur qui provoque ce test est apply. Soit par exemple la procédure :

def (P, proc (k, C1, C2, opérateur))

et l'objet classifié : def (X, as (C, A)).

L'appel de P se fait par :

apply (P, X)

S'il y a correspondance entre C et C1, l'appel généré (pour la compilation ou l'interprétation) est :

opérateur (A)

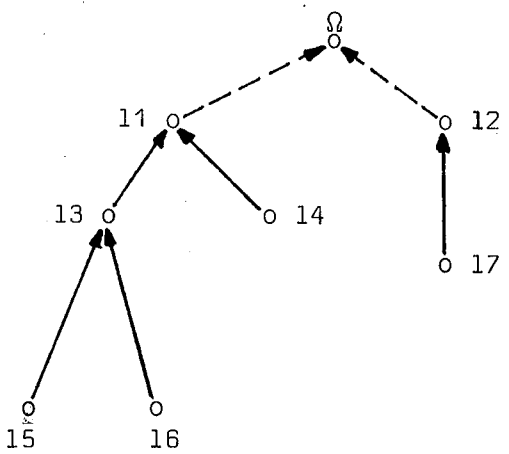
qui est une expression du langage de base dans laquelle les classes ont disparu.

S'il n'y a pas correspondance, une conversion est recherchée entre C et C1, recherche qui nécessite la donnée des graphes (niveaux et classes), le niveau imposé par la procédure k, et un ensemble d'opérations sur les niveaux et les classes que nous allons détailler dans les paragraphes suivants.

4.5.2. Opération sur les niveaux

Les niveaux, ayant une structure d'arbre, sont partiellement ordonnés par la relation i.

Exemple : def (11, level)
def (12, level)
def (13, in 11)
def (14, in 11)
def (15, in 13)
def (16, in 13)
def (17, in 12)



Etant donnés deux niveaux l et l', il n'existe pas toujours d'élément k tel que k = max (l, l') — cas où l appartient à un arbre distinct de l' —. Aussi, on crée un niveau fictif Ω, ce qui revient à changer les définitions :

def (l, level) par def (l, in Ω)

avec la restriction que Ω ne peut être un niveau spécifié dans les conversions ou les procédures.

Avec cette convention, on définit l'opération max de la façon ordinaire par :

max (l1, l2) = l tel que

$[i^+ (l1, l) \wedge i^+ (l2, l)] \wedge$

$[\forall l' \neq l \mid i^+ (l1, l) \wedge i^+ (l2, l')$ on a $i (l, l')]$

où i^+ est la relation i $\cup$ identité.

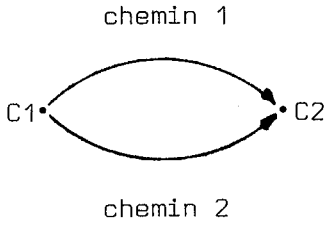
Cette opération est toujours définie.

4.5.3. Opérations sur les relations

Pour sélecter un chemin entre deux classes données, il y a deux opérations à définir :

(1) la comparaison de deux chemins entre deux classes, opération notée +

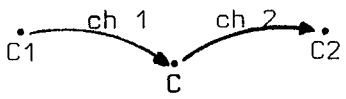
Exemple :



Il faut choisir entre chemin 1 et chemin 2 ; cette opération est commutative, car elle ne dépend pas de l'ordre de comparaison des chemins.

(2) la composition de deux chemins passant par un noeud intermédiaire ; opération notée ×

Exemple :



ch 1 (C1, C) × ch 2 (C, C2) = ch (C1, C2)

Autres notations :

0 (C1, C2) indique l'absence de relations entre C1 et C2

α (C1, C2) indique plusieurs solutions possibles, donc relation ambiguë entre C1 et C2.

Les autres relations considérées sont i et C. Par commodité, on note par les mêmes symboles la relation et la fermeture issue des opérations + et ×.

4.5.3.1 - Opération de comparaison

L'élément neutre est 0 ; l'élément d'absorption α. La table de l'opération est :

+	0	α	i	C (1', L')
0	0	α	i	C (1', L')
α	α	α	α	α
i	i	α	i	i
C (1, L)	C (1, L)	α	i	(1)

Soit k le niveau de la procédure, la case (1) a pour résultat:

C (1, L) + C (1', L') = 1. si 1 = 1' ∧ i (1, k) ⇒ α

2. si i (1, 1') ⇒ 1. i (1', k) ⇒ c (1', L')

2. i (1, k) ∧ i (k, 1')

⇒ C (1, L)

3. $[i^{+}$ (1, 1') ∨ 0 (1, 1')]

∧ [i (k, 1) ∨ 0 (k, 1)] ⇒ 0

4.5.3.2. Opération de composition

L'opération de composition n'est pas commutative. L'élément neutre est i tandis que 0 est élément d'absorption.

On a :

⁂	0	α	i	C (1', L')
0	0	0	0	0
α	0	α	α	α
i	0	α	i	C (1', L')
C (1, L)	0	α	C (1, L)	(2)

La case (2) a pour résultat :

soit m = max (1, 1') ;

C (1, L) ⁂ C (1', L') = 1. si i (m, k) ⇒ C (m, L' • L)

2. si i (k, m) ∨ 0 (k, m) ⇒ 0

où • dénote la composition des opérations de conversion.

4.5.4. Application des conversions

Supposons que, à l'appel apply (P, X), la classe de X (C) ne soit pas celle du paramètre de P qui est C1 ; il y a alors un parcours des chemins de C à C1, par application des opérations + et ⁂ contrôlées par le niveau k, ce qui permet dans certains cas, la détermination de la solution d'une manière plus rapide que si l'on recherchait d'abord le ou les chemins globaux. En fin de calcul, on a une relation entre C et C1 qui donne le résultat d'après le tableau :

Relation	Opération
0 (C, C1)	- Appel incorrect : le test de classe indique qu'il n'y a pas de conversion possible.
α (C, C1)	- Ambiguité de définition : plus d'une conversion possible.
i (C, C1)	as (C2, opérateur (A)) cf. 4.5.1.
C (m, L1 • L2 • ... Ln) (C, C1)	as (C2, opérateur (L1 (L2 ... Ln (A) ...)))

Exemples : En utilisant le graphe de 4.3.3., on peut avoir :

(1) apply (protection de la nature, Fontainebleau)

donne la génération :

op 3 (objet forêt)

à cause de i (forêt, montagne).

(2) apply (chiffre d'affaires, Vanoise)

il y a conversion de parc naturel en station de ski, puis application du membre op 4 de la procédure chiffre d'affaires :

op 4 (op 2 (objet parc naturel))

(3) apply (chiffre d'affaires, Fontainebleau)

il y a ici conversion de déboisement, puis relation i pour donner l'argument du membre op 5 de chiffre d'affaires.

op 5 (op 1 (objet forêt))

CHAPITRE 5

5. DEFINITION FORMELLE

5.1. Présentation de la définition formelle

Inspirée des définitions formelles que plusieurs ont utilisées pour donner la sémantique précise des langages de programmation (Euler [49], EL1 [47] et surtout la méthode de Vienne [46] [28]), celle-ci se divise en trois parties :

(1) syntaxe concrète ;

(2) syntaxe abstraite ;

(3) un interpréteur de la syntaxe abstraite qui définit la sémantique.

La syntaxe concrète est mise sous forme de Backus-Naur [32]. La syntaxe **abstraite** s'inspire de la notation de Van Gils [44] et utilise la description des modes d'Algol 68 à quelques différences près.

L'**interpréteur** est écrit en Algol 68 puisque les arguments de ses procédures sont des catégories syntaxiques, c'est-à-dire, des valeurs de modes définis dans la syntaxe abstraite.

Le processus de traduction de syntaxe concrète à syntaxe abstraite est assez évident, sauf en quelques cas particuliers où la sémantique est complexe. Les catégories syntaxiques deviennent des modes et les règles de grammaire, des déclarations de mode. Les divers éléments d'une partie droite sont groupés sous forme de structure ; les "listes" ou "suites de" donnent des tableaux ; les alternatives sont transformées en unions de mode, avec la différence que les alternatives sont identifiées par des sélecteurs — cf; exemple 3.

Exemples :

(1) Syntaxe concrète :

<unité> ::= (<étiquette>, <defexpr>, <contrôle>)

syntaxe abstraite :

mode unité = struct (etiquette etiq, defexpr defexpr, controle C) ;

(2) Syntaxe concrète :

<programme> ::= <suite de définitions de modules> | <execution>

syntaxe abstraite :

mode programme = union ([ ] definition de module tdfm, execution ex) ;

(3) L'introduction de sélecteurs serait inutile si les modes de l'union étaient tous distincts au sens Algol 68, ce qui n'est pas forcément le cas pour une syntaxe abstraite, d'où les notations de l'interpréteur :

proc ev-prog = (programme prog) :

case prog => tdfm, ex in ....

qui signifie :

Si le programme 'prog' est un tableau de définitions de modules 'tdfm', prendre le premier cas, sinon, si 'prog' est une exécution 'ex', prendre le second cas.

Lorsque les modes de l'union sont distincts, cette écriture est équivalente au test de mode d'Algol 68 :

mode programme = union ([ ] definition de module, execution) ;

proc ev-prog = (programme prog) :

begin [ ] definition de module tdfm, execution ex ;

case tdfm, ex ::= prog in ...

Voir les commentaires (5.4.) pour le choix du symbole =>.

La description formelle est divisée en un certain nombre de parties logiques comprenant chacune les trois composants dont on vient de parler, et un commentaire explicatif. La syntaxe concrète seule est donnée en annexe 2.

5.2. Règles de métasyntaxe

(O-1) <suite de TAG1s> ::= <TAG1> <suite de TAG1s> | <TAG1>

(O-2) <TAG1> ::= <definition de module> | <unité> | <digit> | <car>

(O-3) <liste de TAG2s> ::= (<suite séparée de TAG2s>)

(O-4) <suite séparée de TAG2s> ::= <TAG2>, <suite séparée de TAG2s> |
<TAG2> | ε

(O-5) <TAG2> ::= <index> | <defexpr simple> | <expression> |
<identificateur de mod> | identificateur local |
<definition-c> | <contrôle> | <alternative> |
<interruption>

Les symboles non terminaux sont écrits entre crochets ; les terminaux sont :

- catégories non entre crochets (ex : identificateur local) ;
- suites de caractères soulignés (ex : def, fref) ;
- les symboles délimiteurs : (|) | , | . | * | :

5.3. Définitions préalables pour l'interprétation

L'interpréteur de la syntaxe abstraite utilise des variables globales et des tableaux qui sont définis ici en Algol 68, exemple, le tableau des modules déclarés ou le tableau des index activés. Les paramètres de dimension des tableaux, dépendant des implémentations ne sont pas précisés.

```
int suptindex = co dimension du tableau des index co ;
int nbreindex := 0 ; co nombre courant d'index dans le tableau co
int indice courant ; co indice dans le tableau des index de l'index en
                      cours d'exécution co
[1 : suptindex] struct (ref int ptrunit,
                        ref bool etat,
                        ref bool sema,
                        [ ] ref bool masque,
                        ref locus locex,
                        ref int pty,
                        ref int nbre index ret,
                        [ ] ref index ret) tablindex ;
                    co tableau des index activés co
données modcourant = données of locex of tablindex (indice courant) ;
                    co données significatives pour une exécution co
int ptymin = co priorité minimum co ;
index indexfin = co index spécifique de fin d'exécution tel que
                    actif (indexfin) = true, mais qui n'apparaît pas
                    dans tablindex co ;
[ ] interrupt interst = ((co interruption standard1 co) (co interruption
                        2 co) ...) ;
[ ] interrupt internone = ( ) ;
int suptmod = co dimension du tableau des modules externes co ;
int nbre mod := 0 ; co nombre courant de modules externes co
objet de base nilobj = ''; co objet nul co
[1 : suptmod] struct (idloc idmod, modulelit mod) tmod ;
                co tableau des modules externes co
```

```
modulelit nilmod =
          co module nul co ;
int suptid = co dimension des tableaux des identificateurs dans les
                modules co ;
proc valmod = (idloc id) modulelit :
     co procedure de recherche d'un module externe co
     begin int J ;
        |    for I from 1 to nbremod do
        |    if id = idmod of tmod [I] then J := I ; sortie fi
        |    nilmod • sortie : mod of tmod [J]
      end ;
op belongs to = (idloc nom, [ ] struct (idloc idmod, modulelit mod)
                                tmod) bool :
     co opérateur qui répond vrai si un identificateur appartient à la
        table des modules tmod co
     begin bool b := false ;
        |    for I from 1 to nbre mod do
        |    if b := (nom = id mod of tmod [I]) then sortie fi
        |    sortie : b
      end ;
```

5.4. Structure du programme

Syntaxe concrète :

1 - <programme> ::= <suite de définitions de modules> | <exécution>

2 - <définition de module> ::= def (<identificateur de mod>,
<module litteral>)

3 - <execution> ::= execute (<index>, <module>,
<suite séparée d'expressions>)

4 - <module> ::= <module litteral> | <identificateur>

5 - <unité> ::= (<étiquette>, <defexpr>, <contrôle>)

6 - <étiquette> ::= identificateur local | ε

7 - <identificateur de mod> ::= identificateur local

Syntaxe abstraite et interprétation :

```
mode programme = union ([ ] definition de module tdfm, execution ex) ;
proc ev-prog = (programme prog) :
     case prog ⇒ tdfm, ex in
        for I to upb tdfm do ev-defmod (tdfm [I]),
        ev-execution (ex)
     out erreur syntaxique esac ;
mode definition de module = struct (idmod idmod,
                                    modulelit mod) ;
proc ev-defmod = (definition de module defmod) :
     if idmod of defmod belongs to tmod then double définition de module
     elsf nbre mod = suptmod then depassement capacité tmod
     else nbre mod plus 1 ;
          tmod [nbre mod] := defmod fi ;
mode execution = struct (index index, module mod, [ ] expression parm) ;
proc ev-execution = (execution ex) :
     begin locuslit locexec ;
        | alloc (locexec, act (mod of ex, (false), parm of ex)) ;
        | moniteur (activate : (locexec, index of ex, 'entry', true,
                                interst, ptrmin, (indexfin)))
     end ;
mode module = union (modulelit modlit, identificateur id) ;
```

```
proc ev-module = (module mod) modulelit :
     case mod ->> modlit, id in
          modlit,
          if id belongs to tmod then valmod (idmod)
          else ev-identificateur (id)
     out erreur syntaxique esac ;
mode unité = struct (étiquette etiq, defexpr defexpr, controle c) ;
mode etiquette = idloc ;
mode idmod = idloc ;
```

Commentaires.

L'axiome de la grammaire qui décrit la syntaxe concrète est <programme>. Lorsque l'analyse syntaxique du programme est faite, on obtient un arbre dont la racine est <programme> ; il suffit alors de considérer cet arbre comme une valeur de mode programme et d'appliquer à cette valeur la procédure ev-prog qui interpréte et exécute entièrement le programme donné. Dans le cas d'une règle comportant des alternatives, on comprend mieux la similitude entre le test de conformité de mode (::=) transformé pour notre formalisme en (⇒) et le test de reconnaissance de sous-arbre lors de la transformation d'arbres syntaxiques par macros [36].

Lorsqu'une règle syntaxique a une alternative vide (ε), celle-ci est considérée, dans la syntaxe abstraite comme une valeur particulière du mode qui est défini : exemple pour la règle 6 :

etiquette vide = '' ;

C'est pour cela que l'alternative vide n'apparaît pas dans le mode.

5.5. Evaluation des unités

8 - <defexpr> ::= <liste de defexprs simples> | <defexpr simple> | ε

9 - <defexpr simple> ::= <definition> | <expression>

10 - <expression> ::= <expression> <liste d'expressions> | <objet de base> | <identificateur>

```
mode defexpr = [ ] defexpr simple ;
    proc ev-unité = (defexpr tde) :
        if upb tde = 1 then ev-defexprs (tde [1])
        else defexpr newdefexpr  = tde [1 : upb tde - 1] ;
            (ev-unité (newdefexpr), ev-defexprs (tde [upb tde])) fi ;

mode defexprsimple = union (definition def, expression e) ;
    proc ev-defexprs = (defexprsimple dfs) :
        case dfs => def, e in
            ev-def (def),
            eval (e) out erreur syntaxique esac ;

mode expression = union (struct (expression op, [ ] expression parm) appel,
                                objet de base obj,
                                identificateur id) ;
    proc eval = (expression e) objet de base :
        case e => appel, obj, id in
            eval (op of appel) (evalparm (parm of appel)),
            ev-objet (obj),
            ev-identificateur (id) out erreur syntaxique esac ;
    proc evalparm = ([ ] expression te) [ ] objet de base :
        if upb te = 1 then eval (te [1])
        else [ ] expression newte = te [1 : upb te - 1] ;
            (evalparm (newte), eval (te [upb te])) fi ;
```

L'évaluation des unités conduit à l'évaluation des expressions par la procédure eval et à celle des définitions (cf. 5.6.). Il est à noter que les paramètres d'un appel sont évalués collatéralement et qu'un objet de type opérateur est transformé en procédure et intégré à l'interpréteur par ev-opérateur ce qui étend effectivement cet interpréteur (cf. 5.8.).

5.6. Traitement des identificateurs

11 - <definition> ::= def (identificateur local, <expression>) |
use <liste d'identificateurs de mod>

12 - <identificateur> ::= identificateur local | <identificateur qualifié>

13 - <identificateur qualifié> ::= identificateur local • <identificateur>

```
mode definition = struct (idloc id, expression e) ;

proc ev-def = (definition def) :
    if obj of (tid of modcourant) [valid (id of def, locex of
       tablindex [indice courant])] ≠ nilobj
       then double definition d'identificateur
    elsf nbreid of modcourant = suptid then depassement capacité tid
    else nbreid of modcourant plus 1 ;
         tid [nbreid of modcourant] of modcourant :=
                (id of def, eval (e of def)) fi ;

mode identificateur = union (idloc idloc, idqual idq) ;
mode idqual = struct (idloc idloc, identificateur id) ;
op belongsto = (idloc idloc, locuslit loc) bool :
                valid (idloc, loc) ≠ 0 ;
proc valid = (idloc idloc, locuslit loc) int :
```

```
    begin modulelit modf = ptar of code of loc ;
        int J := 0 ;
        for I from 1 to suptid do
        if idloc = id of (tidf of modf) [I] then J := I ; sortie fi
        sortie : J
    end ;
proc rmod = (idloc idloc, locuslit loc) locuslit :
    if idloc belongsto loc then loc
    elsf code of locenglb of données of loc = nil then identificateur non
                                                        défini
    else rmod (idloc, locenglb of données of loc) fi ;
proc ev-identificateur = (identificateur id) objet de base :
    ev-id (id, locex of tablindex [indice courant]) ;
proc ev-id = (identificateur id, locuslit loc) objet de base :
    case id => idloc, idq in
        (locuslit newloc = rmod (idloc, loc) ;
        obj of (tid of newloc) [valid (idloc, newloc)]),
        ev-idqual (idq, loc)
    out erreur syntaxique esac ;
proc ev-idqual = (idqual idq, locuslit loc) objet de base :
    ev-id (id of idq, ev-id (idloc of idq, loc))
mode idloc = string ;
```

Commentaires.

Le traitement des identificateurs comporte d'une part l'élaboration des définitions (procedure ev-def) et d'autre part la recherche de l'objet — la valeur — correspondant à un identificateur lorsque celui-ci

apparaît dans une expression. On a supprimé l'alternative use de la catégorie 'définition' car, conformément aux remarques de 2.2.3.3., on peut faire une recherche entièrement dynamique en mode interprétatif comme c'est le cas pour la syntaxe abstraite.

La recherche (ev-identificateur) utilise le locus du programme actif (ev-id) pour trouver l'objet soit dans la table courante, soit dans une table des modules englobants. Si l'identificateur est qualifié, il y a recherche à l'aide des valeurs des locus qui font la qualification ;

Exemple : ev-identificateur (A • B • C) =

ev-id (A • B • C, locus courant) =

ev-idqual (A • B • C, locus courant) =

ev-id (B • C, ev-id (A, locus courant)) =

ev-id (A, locus courant) doit donner une valeur de locus : ♯ LA, d'où :

ev-idqual (B • C, ♯ LA) =

ev-id (C, ev-id (B, ♯ LA)) =

ev-id (B, ♯ LA) donne une valeur de locus : ♯ LB

ev-id (C, ♯ LB) qui donne la valeur de l'objet C.

Les autres procédures utilisées sont "valid" qui donne l'indice de l'identificateur dans tid (il va de soi que l'on utilisera en général une méthode plus efficace : dichotomie, hashcoding ...) ; "rmod" fait la recherche de l'identificateur dans les modules englobants.

5.7. Objets de base

14 - <objet de base> ::= <objet litteral> | <objet formel>

15 - <objet litteral> ::= ref (<expression>, <expression>) |
adr (<expression>, <expression>) |
dep (<expression>, <expression>) |
<constante litterale> |
<module litteral> |
<opérateur litteral> | locus | signal

```
16 - <objet formel> ::= fref | fadr | fdep | fconst | fmod | fop |
                        floc | fsig |
17 - <constante litterale> ::= car (<suite de cars>) |
                               bit (<suite de digits>) |
                               hexa (<suite de cars>) |
                               fx (<suite de digits>) |
                               fl (<suite de digits> • <suite de digits>,
                                   <suite de digits>)
18 - <car> ::= <digit> | A | B | C | D  ... Z
19 - <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
mode objet de base = union (objetlit objlit, ref objetlit fobj) ;
mode objetlit = union (struct (expression adr, expression dep) ref,
                       struct (expression ref, expression dep) adr,
                       struct (expression f, expression u) dep,
                       constantelit constlit,
                       modulelit modlit,
                       operateurlit oplit,
                       locus lit loc, signallit sig) ;
mode constantelit = union (string car, [ ] bool bit, string hexa,
                           int fx, real fl) ;
proc ev-objet = (objetlit obj) objetlit :
    case obj => ref, adr, dep, constlit, modlit, oplit, loc, sig in
        (eval (adr of ref), eval (dep of ref)),
        (eval (ref of adr), eval (dep of adr)),
        (eval (f of dep), eval (u of dep)),
        constlit,
        modlit,
        ev-operateur (oplit),
```

loc,

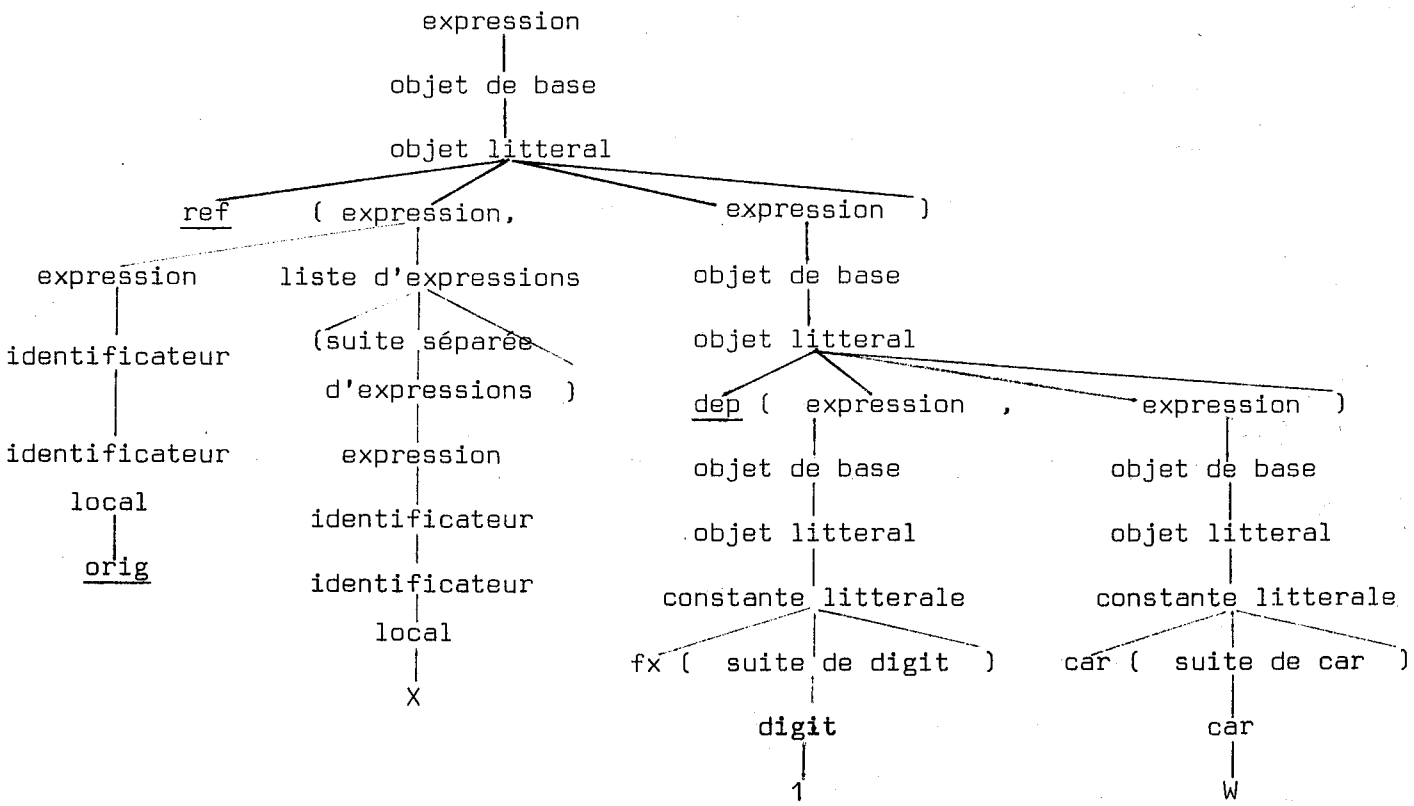
sig

out erreur syntaxique esac ;

Les objets de type données sont des structures (ref, adr, dep) ou des éléments simples (constantes) qui ont une valeur en tant que tels et ne sont pas évalués. La procédure ev-obj évalue les champs des structures. Voici un exemple d'évaluation d'objet de base ; on donne l'arbre syntaxique, puis l'arbre abstrait et enfin l'évaluation.

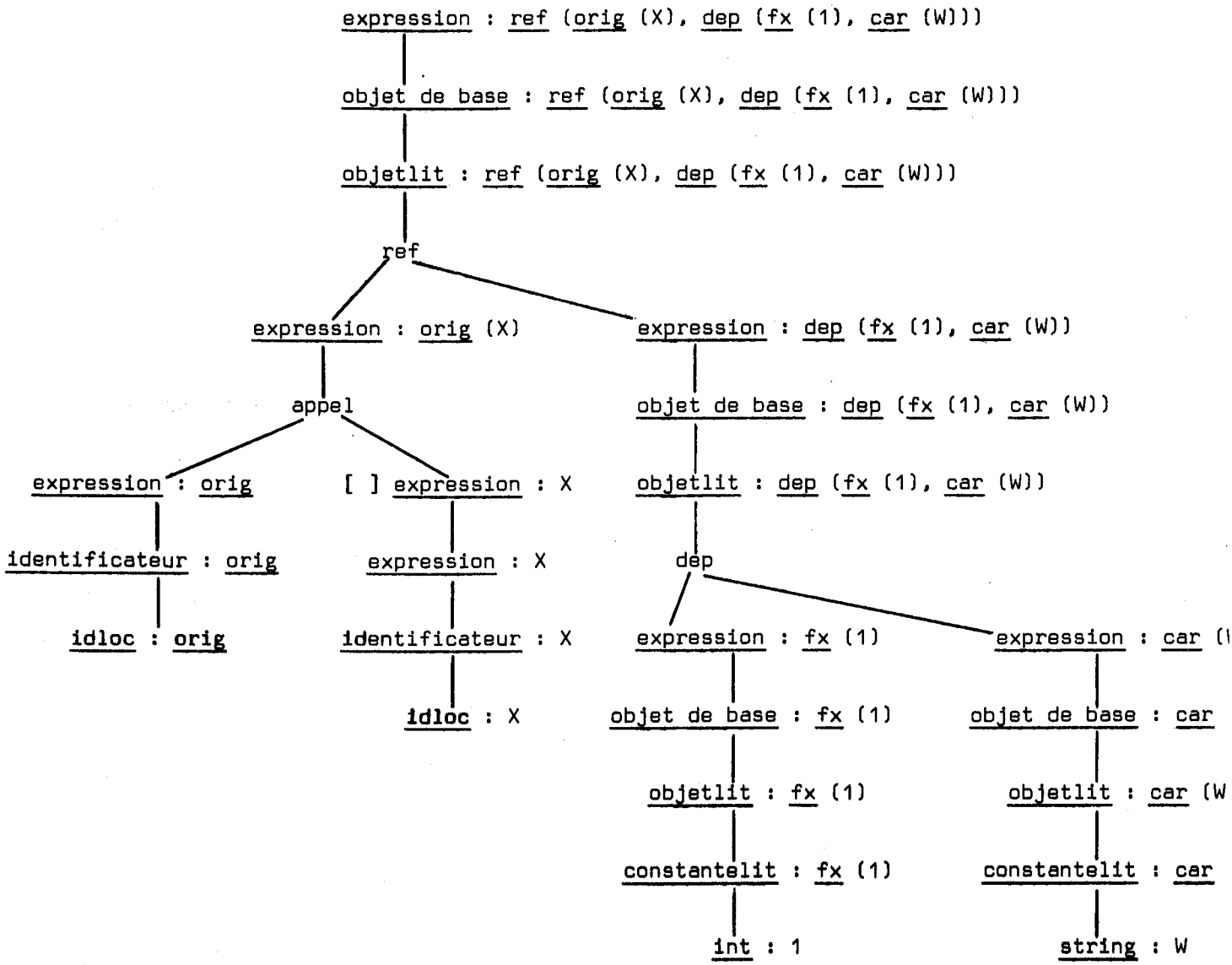
Expression à évaluer : ref (orig (X), dep (fx (1), car (W)))

(F 14) : Arbre syntaxique :



(F 15) : Arbre abstrait.

On a noté chaque noeud par une indication de mode, suivie de la chaîne qui représente la valeur. Certains sélecteurs de modes sont indiqués sur les arcs.



L'évaluation de l'arbre abstrait se fait en appliquant la procédure eval en haut de l'arbre et en développant les appels de procédure d'après des modes rencontrés. On fait le développement des appels en parallèle pour mieux suivre le déroulement dans l'arbre :

eval (ref (orig (X), dep (fx (1), car (W)))) =

ev-objet (ref (orig (X), dep (fx (1), car (W)))) =

(eval (orig (X)), eval (dep (fx (1), car (W)))) =

(eval (orig) (evalparm (X)), ev-objet (dep (fx (1), car (W)))) =

(ev-identificateur (orig) (eval (X)), (eval (fx (1)), eval (car (W)))) =

(ev-identificateur (orig) (ev-identificateur (X)), (ev-objet (fx (1)),
ev-objet (car (W))))=

(ev-identificateur (orig) (ev-identificateur (X)), (1, 'W'))

A ce point de l'évaluation, on retrouve les éléments terminaux de l'arbre abstrait :

(parenthèses de structure de l'objet "ref" ;

ev-identificateur (orig) donne un résultat valeur de l'objet identifié par orig qui doit être un opérateur (cf. 5.8.) ;

(ev-identificateur (X)) paramètre de orig : valeur de l'objet identifié par X qui doit être une référence ;

, séparation des deux champs du "ref" ;

(parenthèse de structure du "dep" ;

1, 'W' champs de "dep" : valeurs constantes ;

) fermeture structure "dep" ;

) fermeture structure "ref".

Remarque : A l'appel ev-objet, le paramètre effectif est un objet de base est le paramètre formel est un objetlit. On n'introduit pas de procédure d'évaluation supplémentaire, car si objet de base est un objetlit, la correspondance se fait normalement comme c'est le cas ici ; s'il s'agit d'un formel (fobj), il y a une conversion de dérepérage pour obtenir un objetlit.

5.8. Objets de base de type programme

20 - <module litteral> ::= mod (<graphe des classes>, <liste d'index>,
<suite d'unités>)

21 - <opérateur littéral> ::= op (corps de base) |
opcomp (<expression>) |
opmod (<expression> , <index>, <result>)

22 - <result> ::= stack <liste d'identificateurs locaux> | ε

```
mode modulelit = struct ([ ] index tindex,
                         [ ] struct (idloc id, bool formel) tidf,
                         ref int nbreid,
                         ref ref modulelit englb,
                         ref int ctrf,
                         ref ref code ptrf,
                         [ ] unité tuf) ;

mode operateurlit = union (op corps,
                           expression opcomp,
                           struct (expression loc, index ind,
                                   result res) opmod) ;

mode result = [ ] idloc ;

mode op = co chaîne de caractères pouvant servir de source à un assembleur
          co ;

mode operateurcalculé = proc ([ ] objetlit) objetlit ;

mode locuslit = struct (ref ref code code, ref ref données données) ;

mode données = struct ([ ] index tindex,
                       [ ] ref struct (bool formel,
                                       objet de base obj) tid,
                       ref int nbreid,
                       ref int ctrd,
```

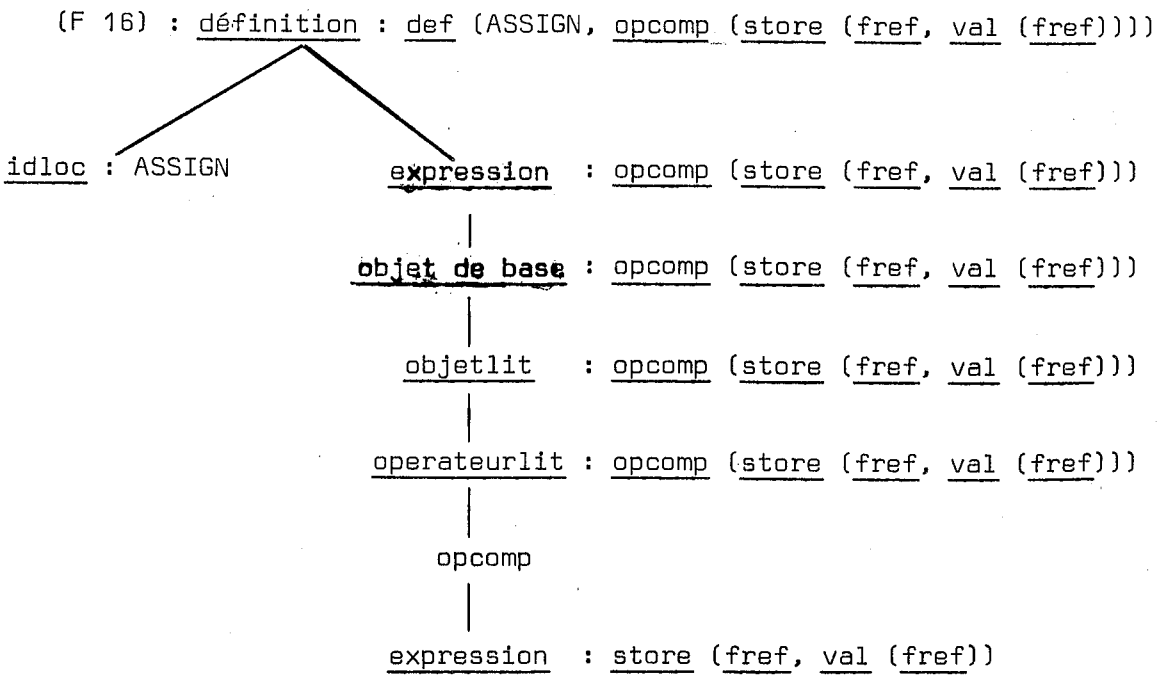
```
                                ref locuslit locenglb) ;
mode code = struct (ref ref modulelit ptrar,
                                [ ] unité tu) ;
proc ev-opérateur = (operateurlit oplit) operateur calculé :
    case oplit => corps, opcomp, opmod in
            co assemblage de la partie corps co,
            co evaluation formelle co,
            co compilation de module co
     out erreur syntaxique esac ;
```

Le principe de l'extension de l'interpréteur prend ici sa forme tangible. Lorsqu'un objet est un "opérateur littéral", il est transformé en "opérateur calculé" et son identificateur devient un point d'entrée de l'interpréteur, dans la portée de cet identificateur.

Considérons l'exemple donné en 2.2.3.9.

```
def (ASSIGN, opcomp (store (fref, val (fref))))
```

L'arbre abstrait de cette définition est :



On peut encore développer l'expression, mais celle-ci n'est pas évaluée par eval, ce qui rend le développement inutile.

L'évaluation formelle :

ev-def (def (ASSIGN, opcomp (store (fref , val (fref))))) =

1 - mise à jour de la table des identificateurs locaux avec l'introduction de "ASSIGN"

2 - évaluation de l'expression correspondant à l'identificateur :

eval (opcomp (store (fref, val (fref)))) =

ev-objet (opcomp (store (fref, val (fref)))) =

ev-operateur (opcomp (store (fref, val (fref))))

L'appel de ev-operateur transforme l'expression formelle :

store (fref, val (fref))

en une proc ([ ] objetlit) objetlit :

et ici plus précisément en proc (ref, ref) : ou "ref" est le premier sélecteur de objetlit. L'intepréteur, qui possédait les entrées store et val a désormais une entrée assign :

- o store ↔ proc (ref, objetlit) :
- o val ↔ proc (ref) objetlit :
- o assign ↔ proc (ref, ref) :

On décrit aussi de la même manière les procédures qui traiten les objets de type programme.

Actualisation : cf. 2.2.3.2.

proc act = (module mod, [ ] bool formel, [ ] expression parm)

locuslit :

```
begin ref données d = heap données ;
      struct (locuslit loc, modulelit mod) actmod = ev-mod (mod) ;
      modulelit modf = mod of actmod ;
      int J := 0 ;
   if ctrf of modf = 0 then
            ref code c = heap code ;
            tu of c := tuf of modf ;
            ptar of c := ref modulelit := modf ;
            ptr of modf := c fi
   tindex of d := tindex of modf ;
   nombreid of d := nombreid of modf ;
   ctrd of d := 0 ;
   ctrf of modf plus 1 ;
   for I to nombreid of d do
   (formel of (tid of d) [I] := formel of (tidf of modf) [I] ;
    obj of (tid of d) [I] := nilobj) ;
    locenglb of d := loc of actmod ;

       Liaison des paramètres :

   for I to nbreid of d do
   if formel of (tid of d) [I] then
         formel of (tid of d) [I] := formel [J plus 1] ;
         obj of (tid of d) [I] := eval (parm [J]) fi ;
   (ptrf of modf, ref ref données := d)
end ;
```

Autres procédures utilisées dans l'actualisation, à savoir "ev-mod" qui appelle "ev-module" (5-4) et "ev-locmod".

```
proc ev-mod = (module mod) struct (locuslit loc, modulelit modf) :
    begin modulelit modbase = ev-module (mod) ;
        (if mod => id Λ id => idqual then ev-locmod (mod)
         elsf englb of modbase = (nil, nil) then (nil, nil)
         else locex of tablindex [indice courant] fi, modbase)
    end ;
proc ev-locmod = (idqual idq) locuslit :
    if id of idq => idloc then ev-identificateur (idloc of idq)
    else ev-locmod (id of idq) fi ;
```

Allocation : cf. 2.2.3.3.

```
proc alloc = (locuslit loc1, locuslit loc2) :
    if loc1 ≠ (nil, nil) then erreur d'allocation else
      ctrd of données of loc2 plus 1 ;
      code of loc1 := code of loc2 ;
      données of loc1 := données of loc2 fi ;
```

Désactualisation : cf. 2.2.3.5.

```
proc desact = (locuslit loc) :
    begin ctrd of données of loc minus 1 ;
        code of loc := nil ;
        données of loc := nil
    end ;
```

Remarque : On suppose que le système récupère la place des zones de données lorsque le "ctrd" de celles-ci est à zéro.

Liaison des paramètres : cf. 2.2.3.4.

```
proc bind = (locuslit loc, [ ] expression parm) :
    begin int J := 0 ;
      |  for I to nbreid of données of loc do
      |    if formel of (tid of données of loc) [I] then
      |      obj of (tid of données of loc) [I] := eval (parm [J plus 1])
      |    fi
    end
```

5.9. Objets utilisés dans le contrôle d'exécution

23 - <index> ::= <identificateur> | * | ret (<index>, <suite séparée d'expression>)

```
mode index = struct (idloc ind, ref int indice) ;
mode signallit = ref [ ] struct (ref locus l,
                                 ref index ind,
                                 ref int ctr) ;
```

Les index d'un module sont reconnus par leur nom ; l'indice est celui du tableau (tablindex) où l'index est représenté quand il est activé. La notation "*" signifie l'index courant, c'est-à-dire celui qui a exécuté l'unité en cours, ou bien le seul index du module dont on parle. La notation "ret" peut être considéré comme un adressage indirect d'index ; c'est plutôt un opérateur qui retourne une valeur d'index.

5.10. Contrôle d'exécution

5.10.1. Syntaxe concrète

24 - <contrôle> ::= <contrôle composé> | <contrôle conditionnel> | <primitive>

25 - <contrôle composé> ::= ctl <liste de contrôles>

26 - <contrôle conditionnel> ::= cond <liste d'alternatives>

27 - <alternative> ::= (<expression>, <contrôle>)

28 - <primitive> ::= activate (<index>, [<paramètres d'activation>]*) |
stop <liste d'index> |
resume <liste d'index> |
kill <liste d'index> |
next (<index>) |
move (<index>, <étiquette>) |
wait (<identificateur>, <liste d'index>, <contrôle>) |
free (<identificateur>, <liste d'index>)

29 - <paramètres d'activation> ::= ENTRY : <etiquette> |
ETAT : <état> | INTERRUPT : <liste d'interruptions> |
PTY : <expression> | RET : <liste d'index>

30 - <etat> ::= PRET | ATTENTE

31 - <interruption> ::= ST | NONE | <type> (<module>, <index>,
<suite séparée d'expressions>)

32 - <type> ::= zerodivide | overflow ...

Pour la règle 28 on a choisi la notation A* où A est une catégorie syntaxique pour désigner l'ensemble :

Si A = x | y | z,

A* = x | y | z | xy | xz | xyz avec ordre des facteurs indifférent.

5.10.2. Syntaxe abstraite du contrôle d'exécution

```
mode contrôle = union (contrôle composé cc, contrôle conditionnel cd,
                       primitive p) ;

mode contrôle composé = [ ] contrôle ;

mode contrôle conditionnel = [ ] struct (expression p, contrôle c) ;

mode primitive = union (activate A, stop S, resume R, kill K, next N,
                        move M, wait W, free F) ;

mode activate = struct (ref locus loc, ref index ind, etiquette entry,
                        bool état,
                        [ ] interrupt int, expression pty,
                        [ ] ref index ret) ;

mode interrupt = struct (locus type, module mod, index ind, [ ] expression
                         parm) ;

mode stop = [ ] ref index ;

mode resume = [ ] ref index ;

mode kill = [ ] ref index ;

mode next = ref index ;

mode move = struct (ref index ind, ref etiquette etiq) ;

mode wait = struct (identificateur idsig, [ ] ref index tind, contrôle c) ;

mode free = struct (identificateur idsig, [ ] ref index tind) ;
```

L'expression p du contrôle conditionnel donne un résultat booléen, d'où son utilisation comme condition.

5.10.3. Moniteur

Le moniteur est l'algorithme qui gère l'exécution des modules. Il est décrit par une procédure. Son premier appel est fait par la procédure "ev-execution" puis il se déroule tant qu'il y a des index dans "tablindex".

```
proc moniteur = (contrôle c) :
    begin bool configuration := false ;
        unité u ;
        ev-contrôle (c) ;
        while nbreindex ≠ 0 do
        (if configuration then indice courant := choix (tablindex) fi ;
        if etat of tablindex [indice courant]
          then u := (tu of code of locex of tablindex [indice courant])
                        [ptrunit of tablindex [indice courant]] ;
               if ¬ sema of tablindex (indice courant)
               then ev-unité (defexpr of u)
               else configuration := true fi
               ev-contrôle (c of u)
          else configuration := true fi)
    end ;
```

La procédure "choix" qui recalcule l'indice courant en fonction des priorités n'est appelée qu'après des fonctions de contrôle qui modifient la configuration : 'activate', 'kill', 'resume', 'free', ou après un 'stop' ou un 'wait' sur l'index qui était précédemment l'index courant.

```
proc choix = (ref [ ] struct ( ... ) tablindex) int :
    begin int pty ; int ptymax = co valeur de la priorité maximum co
        configuration := false ;
        co recherche d'un indice de tableau qui satisfait aux critères
           de priorité et compris entre 1 et nbreindex co
    end ;
```

```
proc ev-contrôle = (contrôle c) :
    case c => cc, cd, p in
        execpar (cc),              co contrôle composé co
        for I to upb cd do         co contrôle conditionnel co
        if eval (p of cd [I]) then ev-contrôle (c of cd [I]) fi,
        execprim (p)               co primitive co
    out erreur syntaxique esac ;

proc execpar = ([ ] contrôle tc) :
    if upb tc = 1 then ev-contrôle (tc [1])
    else [ ] contrôle newte = tc [1 : upb tc-1] ;
        (execpar (newte), ev-contrôle (tc [upb tc])) fi ;

proc execprim = (primitive p) :
    case p => A, S, R, K, N, M, W, F in
                                co traitement d'activation d'un index co
    if ¬ actif (ind of A)
      thef nbreindex = suptindex then dépassement capacité tablindex
          else nbreindex plus 1 ;
               indice of ind of A := nbreindex ;
               ptrunit of tablindex [nbreindex] := unité (entry of A,
                                                   code of loc of A) ;
               etat of tablindex [nbreindex] := etat of A ;
               masque of tablindex [nbreindex] := masquer (int of A) ;
               pty of tablindex [nbreindex] := eval (pty of A) ;
               ret of tablindex [nbreindex] := ret of A ;
               nbreindexret of tablindex [nbreindex] := upb ret of A ;
                  sema of tablindex [nbreindex] := false ;
               locex of tablindex [nbreindex] := loc of A ;
                  configuration := true fi,
```

```
                                    co traitement du stop co
for I to upb S do
   if actif (S [I]) then
      etat of tablindex [indice of S [I]] := false fi,
                                    co traitement du resume co
for I to upb R do
   if actif (R [I]) then
      etat of tablindex [indice of R [I]] := true fi ;
   configuration := true,
                                    co traitement de kill co
for I to upb K do
   if actif (K [I]) then
      if indice of K [I] ≠ nbreindex
      then co soit X l'index tel que indice of X = nbreindex co
          tablindex [indice of X := indice of K [I]] := tablindex [nbreindex]
                                                          fi ;
     nbreindex minus 1 ; indice of K [I] := 0 fi ;
   configuration := true,
                                    co traitement de next co
   if actif (N) thef
     ptrunit of tablindex (indice of N) ≠
             upb (tu of code of locex of tablindex [indice of N])
     then ptrunit of tablindex [indice of N] plus 1
     else execprim (kill : N) fi,
                                    co traitement du move co
  if actif (ind of M) then
     ptrunit of tablindex [indice of ind of M] :=
        unité (etiq of M, code of locex of tablindex [indice of ind of M]) fi
```

```
                                    co traitement du wait co
begin signallit sig = ev-identificateur (idsig of W) ;
   co soit X l'indice dans sig correspondant à l'index courant co
     if ctr of sig [X] > 0 then
        sema of tablindex [indice courant] := true ; sortie
        else sema of tablindex [indice courant] := false ;
            for I to upb tind of W do
               co soit Y l'indice dans sig correspondant à (tind of W) [I] co
               ctr of sig [Y] plus 1 fi ;
     ev-contrôle (c of W) ;
    sortie :
 end,
                                    co traitement du free co
begin signallit sig = ev-identificateur (idsig of F) ;
    for I to upb tind of F do
    co soit Y l'indice dans sig correspondant à (tind of F) [I] co
       ctr of sig [Y] minus 1
 end
out erreur syntaxique esac ;
```

Les commentaires dans wait et free remplacement une procédure qui, étant donné un index et un signal recherche l'occurrence de cet index dans le signal ; s'il ne le trouve pas, il l'insère à la suite en initialisant son compteur, et donne comme résultat l'indice dans le tableau du signal.

```
proc actif = (index I) bool : indice of I ≠ 0 ;
proc unité = (etiquette e, code code) int :
```

```
    begin int J := 0 ;
        for I to upb tu of code do
            if e = etiq of tu [I] then J := I ; sortie fi
            sortie : J
    end ;
proc masquer = ([ ] interrupt int) [ ] bool :
    co mise à jour du masque des interruptions et des locus des routines
       d'interruption co ;
```

Remarque : Après un wait sur lequel l'index courant a été bloqué, l'algorithme de reprise doit être plus fin que celui qui est montré ici, c'est-à-dire que l'on doit essayer de reprendre à l'étape où on s'est arrêté et non au début de l'instruction de contrôle comme semble l'indiquer le moniteur. Ceci a une réelle importance pour les wait imbriqués (cf. exemple 3.5.2.2.).

En conclusion à ce chapitre, je dirai que le formalisme abstrait utilisé ici est simple et relativement aisé à suivre malgré l'aridité de ce genre d'écriture. Le couple syntaxe abstraite - interpréteur définit bien la sémantique à condition que celle du langage de l'interpréteur — ici Algol 68 modifié — soit elle-même parfaitement précise, ce qui est loin d'être évident. Enfin on remarque que les points d'achoppement d'une définition formelle par cette manière sont les identificateurs et le contrôle d'exécution ; mais nous reviendrons là-dessus dans la conclusion.

CHAPITRE VI

6. EXEMPLES D'EXTENSIONS

6.1. Extensions pour l'allocation des variables

Le langage de base ne contient qu'un opérateur pour obtenir une zone libre de la mémoire centrale, c'est "mem", qui est écrit en fonction du système sur lequel le langage doit être implémenté. L'utilisateur a ensuite toute liberté pour organiser l'allocation des données dans cette zone. Mais là se situe un problème dont les solutions ne sont en général que des compromis : dans les langages de haut niveau, on considère que le compilateur prend en charge toute l'allocation et que l'utilisateur n'a pas à s'en soucier, objectif louable mais qui interdit en grande partie ces langages pour l'écriture des systèmes ; d'un autre côté, dans les langages de bas niveau, les lourdeurs de la programmation sont intolérables et encombrent de détails techniques la logique d'un algorithme. La méthode adoptée ici est de laisser la possibilité de gérer l'allocation à son gré en reprenant les opérateurs de base, mais, en plus, de fournir un certain ensemble de définitions et de modules qui peuvent être insérés dans une sorte de "prélude standard" pour obtenir les facilités des langages de haut niveau.

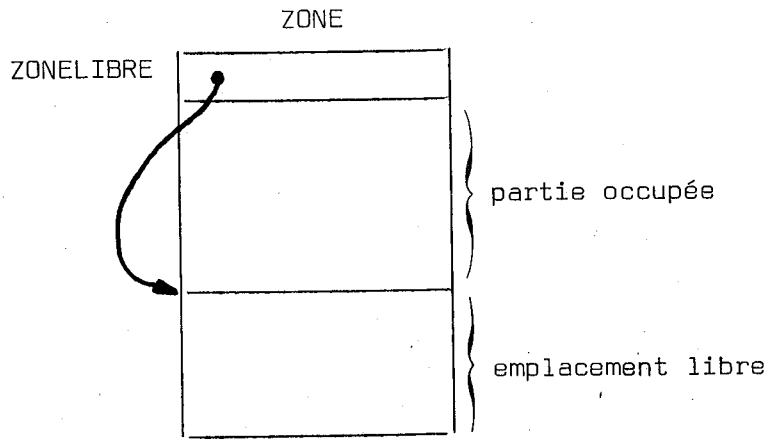
6.1.1. Allocation de variables locales

Le module ALLOCVAR prend en charge l'allocation de variables locales d'une manière contiguë.

```
def (ALLOCVAR, mod (, (I),
        (, def (ZONE, fref), next (*))
        (, def (ZONELIBRE, ref (orig (ZONE), DEPADR (1))), next (*))
        (, store (ZONELIBRE, compadr (orig (ZONE), DEPADR)), next (*))
```

```
(, def (TMOD, mod (, (I),
        (, (def (VAL, fconst), def (CAR, fconst)), next (*))
        (, def (NEWREF, ref (ZONELIBRE, dep (VAL, CAR))), next (*))
        (, store (ZONELIBRE, compadr (val (ZONELIBRE), dep (VAL, CAR))),
                ctl (kill (*), resume (ret (*)))) )), next (*))
(, def (TAILLE, opmod (act (TMOD), I, stack (NEWREF))),
                ctl (kill (*), resume (ret (*))))  ))
```

La configuration en mémoire est la suivante :



Pour pouvoir utiliser les objets ainsi définis, il faut actualiser et exécuter le module de définition :

```
(, (use (ALLOCVAR), def (LOCAV, locus)), next (*))
(, alloc (LOCAV, act (ALLOCVAR, mem (<taille mémoire>))),
        ctl (activate (LOCAV•*), stop (*), next (*)))
```

Il est évident que ces deux unités peuvent être générées par macro, avec une <taille mémoire> totale standard ou passée en paramètre dans la macro.

Dans le reste du programme, on écrit :

```
def (X, taille (1, car (W(2))))
def (Y, taille (1, car (BYTE)))
```

Remarques :

(1) Pour définir le module, on a utilisé l'identificateur DEPADR ; celui-c désigne un déplacement qui correspond au nombre de bits ou de mots nécessaires pour contenir un objet de type adresse. L'implémenteur doi donner la définition de cet identificateur suivant la façon dont il code les objets.

Il le fait par :

def (DEPADR, dep (...))

De même, on a :

def (DEPDEP, dep (...)) - déplacement d'un repère de déplacement ;

def (DEPREF, dep (...)) - déplacement d'un repère de référence.

etc.

D'une manière générale, tous les points qui sont particuliers à la machine ou au système utilisé doivent être définis par l'implémenteur L'annexe III donne les principaux éléments qui dépendent de l'implémentation et sont en quelque sorte les interfaces entre le "hardware" et le "software".

(2) Une extension évidente consiste à transformer par macros la dénotation de chaîne pour arriver à l'écriture habituelle :

car (W) ≡ 'W'

La règle d'extension est :

<constante littérale> → '<suite de cars>' = car (<suite de cars>) ;

6.1.2. Structures

L'écriture des définitions de structures est longue et pénible (cf. 2.2.1.). Une écriture simplifiée est proposée ici par macros syntaxiques du type transformations d'arbres [36]. On propose une description de structure du genre PL/1 ou COBOL avec numéro de niveaux et

indication de la taille de chaque composant.

Exemple :

```
'def( 1 struct 'dep(4,'w'), 2 id1 'dep(1,'w'), 3 id 'dep(1,'byte'),
                                                3 num 'dep(3,'byte'),
                                    2 nom 'dep(2,'w'),
                                    2 suivant 'dep(1,'w'))
```

L'écriture en langage de base est une définition collatérale :

```
( 'DEF( STRUCT , TAILLE ( 4 , 'W')), 'DEF( ID1 , 'REF( ORIG ( STRUCT ),
 'DEP( 1 , 'W'))), 'DEF( ID , 'REF( ORIG ( ID1 ), 'DEP( 1 , 'BYTE'))),
 'DEF( NUM , 'REF( 'ADR( ID1 , 'DEP( 1 , 'BYTE')), 'DEP( 3 , 'BYTE'))),
 'DEF( NOM , 'REF( 'ADR( STRUCT , 'DEP( 1 , 'W')), 'DEP( 2 , 'W'))),
 'DEF( SUIVANT , 'REF( 'ADR( STRUCT , 'DEP( 3 , 'W')), 'DEP( 1 , 'W')))
)
```

La transformation du langage étendu en langage de base a été faite automatiquement par le macro-processeur de Ph. CHATELIN [8] qui sert d'outil pour toutes les extensions syntaxiques. Les règles de transformations sont données ci-après. Elles comprennent un sous-ensemble des règles de la syntaxe de base (Annexe 2), les deux macros de transformation (algorithme p. 108), et des fonctions sémantiques du macro-processeur comme des opérations sur les entiers et l'impression d'un message. On donne en Annexe 4 une présentation succinte du macro-processeur qui permet de comprendre le formalisme utilisé. Les symboles précédés de ' représentent les unités syntaxiques ('def, 'ref, ...).

Règles de transformation des structures :

```
ST -> '''DEF(' SDF ')' = '(' SDF ')' | '(' SDF ')' ;
PILE -> 'P(' DIG ID DEP ')' | PILE 'P(' DIG ID DEP ')' ;
SDF -> DF | SDF ',' DF ;
DF -> '''DEF(' ID ',' EXP ')' ;
EXP -> EXP '(' SSEXP ')' | OB | ID;
SSEXP -> EXP | SSEXP ',' EXP ;
OB -> '''REF(' EXP.1 ',' EXP.2 ')' | '''ADR(' EXP.1 ',' EXP.2 ')' |
          DEP | CAR | DIG ;
DEP -> '''DEP(' DIG ',' CAR ')' ;
CAR -> '''B''' | '''BYTE''' | '''HW''' | '''W''' | '''DW''' ;
STRUCT -> DIG ID DEP | ',' DIG ID DEP | ')' ;
X -> 'X(' DIG ID.1 ID.2 DEP.1 DEP.2 ')' | () ;

SDF -> STRUCT =
     (STRUCT -> DIG.1 ID DEP |(DIG.2:COPIE(DIG.1)
     |(DECR(DIG.2) |(ISZERO(DIG.2)|(DEP -> '''DEP(' DIG.3 ',' CAR ')' |
      '''DEF(' ID ',TAILLE(' DIG.3 ',' CAR '));;P(' DIG.1 ID DEP ')' )|
     (MESSAGE('PREMIER NIVEAU NON 1')|¬))))|
     (STRUCT -> ',' DIG ID DEP | DIG ID DEP |=));

SDF ->  SDF ';' X ';' PILE.1 STRUCT =
     (STRUCT -> ')' | SDF ')' |
     (STRUCT -> DIG ID DEP | SDF ';' X ';' PILE.1 ',' DIG ID DEP |
     (STRUCT -> ',' DIG.1 ID.1 DEP.2 |
     (DEP.2 -> '''DEP(' DIG.2 ',' CAR.2 ')' |
     (PILE.1 -> PILE.2 'P(' DIG.3 ID.2 DEP.4 ')' |
     ( DEP.4 -> '''DEP(' DIG.4 ',' CAR.4 ')' |
     (X -> () |(INF(DIG.3 , DIG.1) |
      SDF ','''DEF(' ID.1 ','''REF(ORIG(' ID.2 '),''DEP(' DIG.2 ','
     CAR.2 '')));' X ';' PILE.1 'P(' DIG.1 ID.1 DEP.2 ')' |
     (EGAL(DIG.3 , DIG.1) |
      SDF ';X(' DIG.3 ID.1 ID.2 DEP.4 DEP.2 ');' PILE.2 STRUCT |
      SDF ';' X ';' PILE.2 STRUCT )) |
     (X -> 'X(' DIG.5 ID.3 ID.4 DEP.6 DEP.7 ')' |
     (DEP.6 -> '''DEP(' DIG.6 ',' CAR.6 ')' |
     ( DEP.7 -> '''DEP(' DIG.7 ',' CAR.7 ')' |
     ( DIG.9 : ADD(DIG.6 , DIG.7) |
      SDF ','''DEF(' ID.3 ','''REF(''ADR(' ID.2 ','''DEP(' DIG.6 ','
     CAR.6 ')),''DEP(' DIG.7 ',' CAR.7 '));;'
     PILE.1 'P(' DIG.5 ID.4 '''DEP(' DIG.9 ',' CAR.6 '))' |¬)))))) |
     (PILE.1 -> 'P(' DIG.3 ID.2 DEP.4 ')' |
     ( DEP.4 -> '''DEP(' DIG.4 ',' CAR.4 ')' |
     ( X -> 'X(' DIG.5 ID.3 ID.4 DEP.6 DEP.7 ')' |
     ( DEP.6 -> '''DEP(' DIG.6 ',' CAR.6 ')' |
     ( DEP.7 -> '''DEP(' DIG.7 ',' CAR.7 ')' |
     ( DIG.9 : ADD(DIG.6 , DIG.7) |
      SDF ','''DEF(' ID.3 ','''REF(''ADR(' ID.2 ','''DEP(' DIG.6 ','
     CAR.6 ')),''DEP(' DIG.7 ',' CAR.7 '));;'
     PILE.1 'P(' DIG.5 ID.4 '''DEP(' DIG.9 ',' CAR.6 '))' |¬))|
     (INF(DIG.3 , DIG.1) |
      SDF ','''DEF(' ID.1 ','''REF(ORIG(' ID.2 '),''DEP('
     DIG.2 ',' CAR.2 '));' X ';' PILE.1 'P(' DIG.1 ID.1 DEP.2 ')' |
     (MESSAGE('DEFINITION DE STRUCTURE INCORRECTE')|¬)))))))))));
ID -> *L1; DIG -> *L2;
EGAL -> *F9;DECR -> *F6;ISZERO -> *F7;COPIE -> *F8;
INF -> *F10;ADD -> *F11;MESSAGE -> *F5;
```

Algorithme des règles de transformation des structures -

La première règle de transformation :
"SDF → STRUCT =" s'applique à la structure majeure et teste si le premier niveau est "1". Elle construit une pile qui contiendra les informations sur les niveaux qui ont déjà été traités ; elle génère une définition avec l'opérateur taille.

La deuxième règle : "SDF → SDF ';' X ';' PILE•1 STRUCT =" teste le numéro de la sous-structure nouvelle avec le numéro de niveau du sommet de pile. Trois cas se présentent :

1 - le numéro de la pile est inférieur au numéro de la structure, c'est-à-dire que l'on rentre dans une sous-structure inférieure ; il y a alors génération d'une définition avec l'opérateur orig et mise dans la pile des caractéristiques de cette sous-structure.

2 - le numéro de la structure est égal au numéro de pile ; on reste dans le même niveau ; il y a génération d'une définition avec un calcul d'adresse (objet adr) et mise à jour du sommet de pile par l'intermédiaire de la catégorie X.

3 - le numéro de la structure est inférieur au numéro de la pile ; on sort d'une sous-structure. Cela revient à enlever le sommet de pile et à recommencer les tests avec le nouveau sommet.

Cette syntaxe permet de faire certaines vérifications ; on doit dire cependant que les problèmes de cadrage ne sont pas tous entièrement résolus.

6.1.3. Allocation des tableaux

Par une méthode analogue à l'allocation des variables (6.1.1.), on peut introduire l'allocation et la recherche des éléments de tableaux.

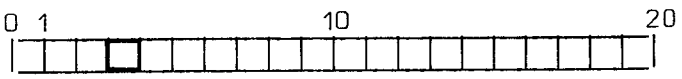
- Soit à définir un tableau à une dimension, de vingt éléments, chaque élément ayant la taille d'un mot :

```
(, def (TAB, taille (20, 'W')), next (*))
(, def (ACCESMOD, mod (, (I),
      (, (def (T, fref), def (INDICE, fconst),
          def (TAILLELEM, fdep),
          def (ELEM, taille (f (DEPREF), u (DEPREF)))), next (*))
      (, store (ELEM, ref (adr (T, dep (mult (INDICE, f (TAILLELEM)),
                                        u (TAILLELEM))),
                               TAILLELEM)), ctl (kill (*), resume (ret (*)))) )),
                                        next (*))
(, def (ACCES, opmod (act (ACCESMOD), I, stack (ELEM))), ...
```

A l'aide de ces définitions, on peut faire appel à ACCES pour obtenir une "tranche" de tableau :

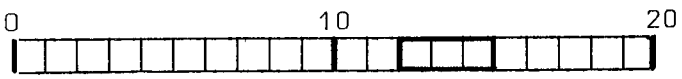
```
acces (TAB, 3, dep (1, 'W'))
```

donne la référence




```
acces (TAB, 4, dep (3, 'W'))
```

donne dans le même tableau :



On peut lier plusieurs paramètres dans l'opérateur :

```
def (ACCESTAB, opmod (act (ACCESMOD, TAB,, dep (1, 'W')), I,
                      stack (ELEM)))
```

Alors, l'appel : accestab (6)

donne sur la pile d'évaluation le 6ème élément du tableau (en commençant le compte des éléments à zéro).

Une autre façon de faire est d'utiliser la définition de ACCESMOD dans une macro de définition de tableaux :

- Soit l'équivalence :

(<e>, def (<id>, tableau (<dim>, <facteur>, <déplacement>)), <contrôle>)

≡ (<e>, def (<id> REF, taille (mult (<dim>, <facteur>), <déplacement>)),
next (*))

(, def (<id>, opmod (act (ACCESMOD, <id> REF,,
dep (<facteur>, <déplacement>)),
*, stack (ELEM))), <contrôle>)

La définition du tableau TAB revient à écrire :

def (TAB, tableau (20, 1, 'W'))

où 20 est le nombre d'éléments
1, 'W', les paramètres de l'encombrement de chacun.

L'accès à un élément de tableau est :

tab (6) qui est une référence.

On peut écrire donc : assign (tab (2), tab (4))

Remarque : La définition de ACCESMOD donne l'algorithme de recherche des éléments d'un tableau. Il est simple, sur ce modèle, d'écrire les modules pour l'accès des tableaux à plusieurs dimensions, des tableaux diagonaux, etc....

6.2. Extension par actualisation

On a utilisé l'actualisation d'un module pour l'utilisation de tableaux (6.1.3.). En règle générale, on peut avoir recours à un module lorsqu'on doit exécuter certains calculs ; c'est le cas de la boucle 'pour' que l'on peut définir de deux façons.

6.2.1. Système de macros simple

On donne à la page suivante la grammaire du langage de base sous forme hors-contexte, issue de l'annexe 2. A la fin de cette grammaire se trouvent un certain nombre de macros (cf. Annexe 4) qui modifient l'écriture des unités, et introduisent l'expression conditionnelle "si-alors-sinon", le "allera", ainsi que la boucle "pour" dont le remplacement est assez simple à comprendre. La fonction sémantique "new" permet de générer les étiquettes.

```
PROG -> SDFM | EX ;
SDFM -> DFM | DFM SDFM ;
DFM -> '''DEF(' IDM ',' ML ')' ;
EX -> '''EXECUTE(' I ',' M ',' SSEXP ')' ;
M -> ML | IDM ;
E -> IDLOC | () ;
IDM -> IDLOC ;
DE -> LDES | DES | () ;
LDES -> '(' SSDES ')' ;
SSDES -> DES | DES ',' SSDES ;
DES -> DF | EXP ;
EXP -> EXP LEXP | OB | ID ;
LEXP -> '(' SSEXP ')' ;
SSEXP -> EXP | EXP ',' SSEXP | () ;
DF -> '''DEF(' IDLOC ',' EXP ')' | '''USE(' SSIDM ')' ;
SSIDM -> IDM | IDM ',' SSIDM ;
ID -> IDLOC | IDQ ;
IDQ -> IDLOC '.' ID ;
OB -> OBL | OBF ;
OBL -> '''REF(' EXP ',' EXP ')' | '''ADR(' EXP ',' EXP ')' |
     '''DEP('EXP ',' EXP ')' | CL | ML | OPL | '''LOCUS' |
     '''SIGNAL' ;
OBF -> '''FREF' | '''FADR' | '''FDEP' | '''FCONST' | '''FMOD' |
     '''FOP' | '''FLOC' | '''FSIG' ;
ML -> '''MOD(' LI ',' SU ')' ;
OPL -> '''OP(' CB ')' | '''OPCOMP(' EXP ')' | '''OPMOD(' EXP ',' I ','
     RES ')' ;
RES -> '''STACK(' SSIDLOC ')' | () ;
SSIDLOC -> IDLOC | IDLOC ',' SSIDLOC ;
LI -> '(' SSI ')' ;
SSI -> I | I ',' SSI | () ;
SU -> U | U SU ;
CL -> '''CAR(' CAR ')' | '''BIT(' DIG ')' | '''HEXA(' CAR ')' |
     '''FX(' DIG ')' |
     '''FL(' DIG '.' DIG ',' DIG ')' | DIG ;
I -> ID | '*' | 'RET(' I ',' SSEXP ')' | 'RET(' I ')' | ID '.*' ;
C -> CP | CC | P =(P -> () | 'NEXT(*)' |=) ;
CP -> 'CTL(' SSC ')' ;
SSC -> C | C ',' SSC ;
CC -> 'COND(' SSA ')' ;
SSA -> A | A ',' SSA | () ;
A -> '(' EXP ',' C ')' ;
P -> 'ACTIVATE(' I ',' PAV ')' | 'STOP' LI | 'KILL' LI | 'RESUME' LI |
     'NEXT(' I ')' | 'MOVE(' I ',' E ')' | 'WAIT(' ID ',' LI ',' C ')'
     | 'FREE(' ID ',' LI ')' |'ACTIVATE(' I ')' ;
PAV -> PAV ',' PA | PA ;
PA -> 'ENTRY:' E | 'ETAT:' ET | 'INTERRUPT:(' INT ')' | 'PTY:' EXP |
     'RET:' LI ;
ET -> 'PRET' | 'ATTENTE' ;
INT -> 'ST' | 'NONE' | T '(' M ',' I ',' SSEXP ')' ;
```

```
T -> 'ZERODIVIDE' | 'OVERFLOW' ;
CAR -> *L1 ;
CB -> *L1 ;
IDLOC -> *L1 ;
DIG -> *L2 ;
PROG -> '''PROG' SU '''FPROG' = '''EXECUTE(I,''MOD((I),' SU '),)' ;
U -> E ':' DE C ';' = '(' E ',' DE ',' C ')'|
     DE C ';' = '(,' DE ',' C ')' ;
CC -> 'SI' EXP 'ALORS' C.1 'SINON' C.2 'FINSI' =
     'COND((' EXP ',' C.1 '),(TRUE,' C.2 '))' ;
P -> 'ALLERA' E  = 'MOVE(*,' E ')' ;
P -> ')' = 'NEXT(*))' |';' = 'NEXT(*);' |',' = 'NEXT(*),' ;
CP->'RETURN' = 'CTL(KILL(*),RESUME(RET(*)))' ;
DE -> '''POUR' EXP.1 '''DE' EXP.2 '''PAS' EXP.3 '''A' EXP.4
     '''TANTQUE' EXP.5 '''FAIRE' DE.1 ;
U -> '(' E.1 ',' DE.1 ',' C.1 ')' =
     (DE.1 -> '''POUR' EXP.1 '''DE' EXP.2 '''PAS' EXP.3
          '''A' EXP.4 '''TANTQUE' EXP.5 '''FAIRE' DE.2 |
          (IDLOC.1: NEW |(IDLOC.2: NEW |
 '(' E.1 ',STORE(' EXP.1 ',' EXP.2'),NEXT(*))'
 '(' IDLOC.1 ',,COND((OU(SUP(VAL(' EXP.1 '),' EXP.4 '),NON(' EXP.5')),'
                    'MOVE(*,' IDLOC.2 ')),(TRUE,NEXT(*))))'
 '(,' DE.2 ',NEXT(*))'
 '(,STORE(' EXP.1 ',PLUS(VAL(' EXP.1 '),' EXP.3 ')),MOVE(*,'IDLOC.1'))'
 '(' IDLOC.2 ',,' C.1 ')' |¬)|¬)|=)  ;
NEW -> *F2 ;
```

Cette grammaire a permis d'analyser et de transformer en langage de base la définition du module Pascal qui initialise une matrice triangulaire inférieure avec les valeurs du triangle de Pascal. Ce module utilise la fonction acces (6.1.3) pour définir l'opérateur elem qui donne un élément du tableau défini par deux indices, le premier (I) indiquant la ligne, le deuxième (J), la colonne. Comme elem n'est utilisé que dans ce module, on ne vérifie pas si J ≤ I. Ce module contient deux paramètres : la référence définissant le tableau, et un entier indiquant le nombre de lignes.

```
'DEF(PASCAL,'MOD((INDEX),
     ('DEF(A,'FREF),'DEF(N,'FCONST));
     ('DEF(I,TAILLE(1,'CAR(W))),'DEF(J,TAILLE(1,'CAR(W))));
     'DEF(T,BIT(1));
     'DEF(ELEM,'OPMOD(ACT('MOD((INDEX2),
          'DEF(I ,'FCONST);
          'DEF(R,ACCES(A,PLUS(DIV(MULT(MOINS(I,1),I),2),'FCONST),
                  'DEP(1,'CAR(W)))) RETURN; )),
          INDEX2,'STACK(R)));
   INIT:
     'POUR I 'DE 1 'PAS 1 'A N 'TANTQUE T 'FAIRE
          (STORE(ELEM(VAL(I),1),1),
           STORE(ELEM(VAL(I),VAL(I)),1));
   CALCUL:
     'POUR I 'DE 3 'PAS 1 'A N 'TANTQUE T 'FAIRE
     'POUR J 'DE 2 'PAS 1 'A MOINS(VAL(I),1) 'TANTQUE T 'FAIRE
          STORE(ELEM(VAL(I),VAL(J)),
                PLUS(VAL(ELEM(MOINS(VAL(I),1),MOINS(VAL(J),1))),
                     VAL(ELEM(MOINS(VAL(I),1),VAL(J)))))
     RETURN ; ))
```

La génération du module en langage de base est :

```
 'DEF( PASCAL , 'MOD(( INDEX ),(,( 'DEF( A , 'FREF), 'DEF( N , 'FCONST)
), NEXT(*))(,( 'DEF( I , TAILLE ( 1 , 'CAR( W ))), 'DEF( J , TAILLE (
 1 , 'CAR( W )))), NEXT(*))(, 'DEF( T , BIT ( 1 )), NEXT(*))(, 'DEF(
 ELEM , 'OPMOD( ACT ( 'MOD(( INDEX2 ),(, 'DEF( I , 'FCONST), NEXT(*))(
, 'DEF( R , ACCES ( A , PLUS ( DIV ( MULT ( MOINS ( I , 1 ), I ), 2 ),
 'FCONST), 'DEP( 1 , 'CAR( W )))), CTL( KILL(*), RESUME( RET(*)))))),
 INDEX2 , 'STACK( R )), NEXT(*))( INIT , STORE ( I , 1 ), NEXT(*))(
 @3  ,, COND(( OU ( SUP ( VAL ( I ), N ), NON ( T )), MOVE(*, @4  )),(
 TRUE , NEXT(*))))(,( STORE ( ELEM ( VAL ( I ), 1 ), 1 ), STORE ( ELEM
( VAL ( I ), VAL ( I )), 1 )), NEXT(*))(, STORE ( I , PLUS ( VAL ( I ),
 1 )), MOVE(*, @3  ))( @4  ,, NEXT(*))( CALCUL , STORE ( I , 3 ), NEXT(
*))( @5  ,, COND(( OU ( SUP ( VAL ( I ), N ), NON ( T )), MOVE(*, @6  )
),( TRUE , NEXT(*))))(, STORE ( J , 2 ), NEXT(*))( @7  ,, COND(( OU (
 SUP ( VAL ( J ), MOINS ( VAL ( I ), 1 )), NON ( T )), MOVE(*, @8  )),(
 TRUE , NEXT(*))))(, STORE ( ELEM ( VAL ( I ), VAL ( J )), PLUS ( VAL (
 ELEM ( MOINS ( VAL ( I ), 1 ), MOINS ( VAL ( J ), 1 ))), VAL ( ELEM (
 MOINS ( VAL ( I ), 1 ), VAL ( J ))))), NEXT(*))(, STORE ( J , PLUS (
 VAL ( J ), 1 )), MOVE(*, @7  ))( @8  ,, NEXT(*))(, STORE ( I , PLUS (
 VAL ( I ), 1 )), MOVE(*, @5  ))( @6  ,, CTL( KILL(*), RESUME( RET(*)))
)))
```

6.2.2. Actualisation et macros syntaxiques

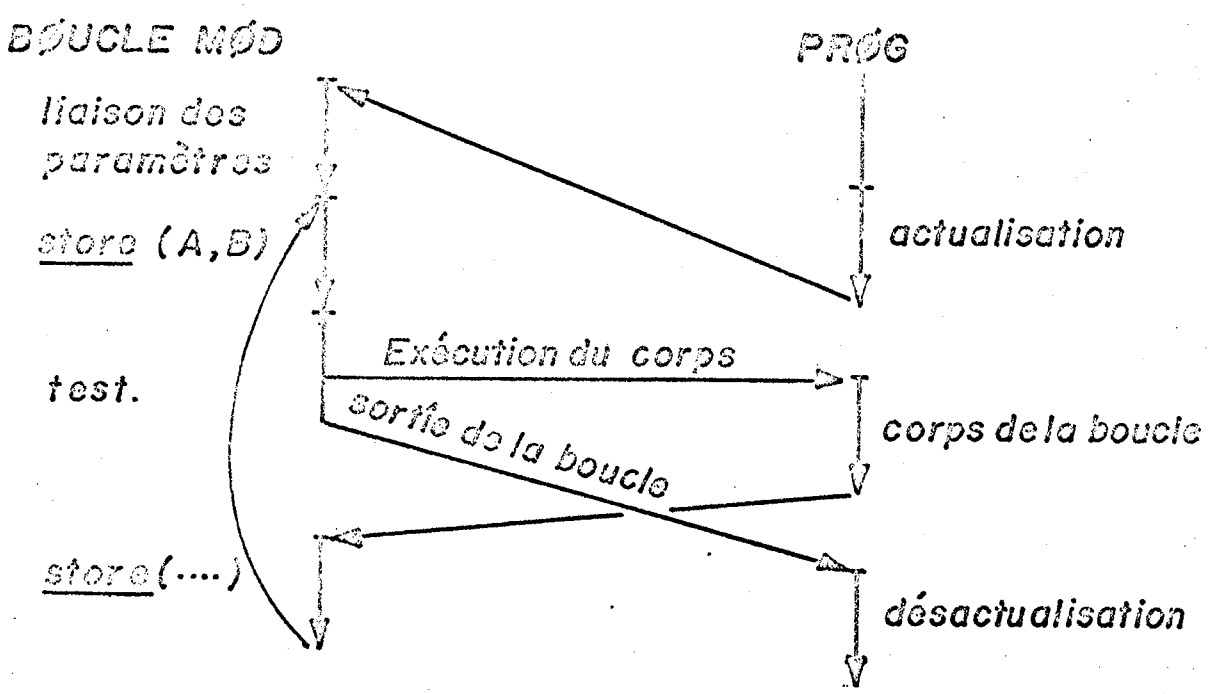
```
def (BOUCLEMOD, mod (, (I),
    (, (def (A, fref), def (B, fconst), def (C, fconst),
        def (D, fconst), def (P, fconst)), next (*))
    (L1, store (A, B), next (*))
    (,, cond (( ou (sup (val (A), C), non (P)),
                        ctl (next (ret (*)), kill (*)), resume (ret (*)))),
            (true, ctl (next (*), stop (*), resume (ret (*))))))
    (, store (A, plus (val (A), D)), move (*, L1))))
```

L'appel de la boucle peut être, en écriture de base :

```
(, def (BOUCLE, locus), next (*))
(, alloc (BOUCLE, act (BOUCLEMOD, R, 1, 3, 100, true)),
        ctl (next (*), stop (*), activate (BOUCLE • *)))
(, <corps de la boucle>, resume (BOUCLE • *))
(, desact (BOUCLE), ....
```

Le passage de contrôle entre le programme principal et le sous-programme constitué par le module BOUCLEMOD est représenté par le schéma : (F 17).

(F 17)



L'appel de BOUCLEMOD peut être généré par une macro attaché à la règle de la boucle "pour". En utilisant la grammaire donnée en 6.2 il suffit de modifier la règle correspondante par la macro :

```
U -> '(' E.1 ',' DE.1 ',' C.1 ')' =
     (DE.1 -> '''POUR' EXP.1 '''DE' EXP.2 '''PAS' EXP.3
          '''A' EXP.4 '''TANTQUE' EXP.5 '''FAIRE' DE.2 |
          (IDLOC.1:NEW|
     '(' E.1',''DEF(' IDLOC.1 ','LOCUS),NEXT(*))'
     '(,ALLOC(' IDLOC.1 ',ACT(BOUCLEMOD,' EXP.1 ',' EXP.2 ',' EXP.3 ','
          EXP.4 ',' EXP.5 ')),'
          'CTL(NEXT(*),STOP(*),ACTIVATE(' IDLOC.1 '.*)))'
     '(,' DE.2 ',RESUME(' IDLOC.1 '.*))'
     '(,DESACT(' IDLOC.1 '),' C.1 ')'|¬)|=) ;
```

On donne un petit exemple de boucle transformée de cette manière :

```
'prog exemple:
     'pour x 'de 1 'pas 4 'a 20 'tantque sup(a,x)
     'faire store(a,moins(y,x))
     return ;
'fprog
```

Le résultat du remplacement est le programme en langage de base :

```
'EXECUTE( I , 'MOD(( I ),( EXEMPLE , 'DEF( @2  , 'LOCUS), NEXT(*))(,
ALLOC ( @2  , ACT ( BOUCLEMOD , X , 1 , 4 , 20 , SUP ( A , X ))), CTL(
NEXT(*), STOP(*), ACTIVATE( @2   .*)))(, STORE ( A , MOINS ( Y , X )),
RESUME( @2   .*))(, DESACT ( @2  ), CTL( KILL(*), RESUME( RET(*)))),)
```

6.3. Dialecte à structure de bloc

La façon d'introduire un bloc a été présentée en 3.4.8. Nous allons ici développer des extensions sur ce thème.

6.3.1 - Blocs

Nous reprenons l'exemple donné au chapitre 3 :

```
def (PROG, mod (, (I),
        .
        .
        .
      (, def (B1, locus), next (*))
      (, alloc (B1, act (mod (, (J),
              (, def (N, ref (...)), next (*))      }
                .                                   }
                .                                   }  Bloc B1.
                .                                   }
              (, ——, ctl (kill (*), resume (ret (*)))))  }
              )), ctl (stop (*), next (*), activate (B1 • *)))
      (, desact (B1), next (*))
```

En synthétisant les opérations à effectuer lors de l'ouverture et de la fermeture d'un bloc, on a les équivalents qui peuvent s'écrire par macros :

```
B1 : begin ; <=> { (, def (B1, locus), next (*))
                 { (, alloc (B1, act (mod (, (J),

end ;       <=> { (,, ctl (kill (*), resume (ret (*)))))
                 { )), ctl (stop (*), next (*), activate (B1 • *)))
                 { (, desact (B1), next (*))
```

L'étiquette du bloc, qui devient un identificateur de locus peut être générée par la macro.

Moyennant cette extension, les blocs apparaissent naturellement :

```
def (ID, mod (, (I),
     (, def (A, taille (1, 'W')), ...
       .
       .
       .
     B1 : begin ;
     |   (, def (C, taille (1, 'W')), ...
     |     .
     |     .
     |     .
     |   B2 : begin ;
     |   |   (, def (A, taille (2, 'BYTE')), ...
     |   |     .
     |   |     .
     |   |     .
     |   end ;
     |     .
     |     .
     |     .
     end ;
          .
          .
          .
           ))
```

La portée des identificateurs est celle d'Algol 60, c'est-à-dire que le premier identificateur A est connu dans le programme au niveau 0, ainsi que dans le bloc B1, mais non dans le bloc B2 où il est redéfini ; C est connu dans B1 et B2 et le deuxième identificateur A dans le bloc B2 uniquement.

La zone de données d'un module interne contient une mémoire pour le locus du module englobant qui est forcément actualisé. En remontant ainsi dans la chaîne des modules englobants, on retrouve les variables qui sont globales au bloc considéré (cf. 5.8., la description du mode "données").

6.3.2. Etude des chaînages

Pour bien comprendre les détails des chaînages et leur réalisation dans l'implémentation du langage de base, nous allons traiter complètement, de ce point de vue, un exemple comprenant des modules internes formant blocs, et d'autres formant des procédures (sous-programmes - cf. 3.4.6.) récursives. Le programme est d'abord donné dans un style Algol assez libre, dans lequel on ne s'intéresse qu'aux déclarations et aux appels ; les unités de calcul ne sont pas décrites.

```
PROG : programme ;
    |   ref A ;
    |   B1 : begin ;
    |     |   ref N ;
    |     |   proc P = (formel X ;
    |     |             |   B2 : begin ;
    |     |             |     |   ref Q ;
    |     |             |     |   ref L ;
    |     |             |     |      .
    |     |             |     |      .
    |     |             |     |      .
    |     |             |     |   if L = 0 then call P(Q) ;
    |     |             |   end ;
    |     |             |     |_____niveau 3
    |     |             )_____niveau 2
    |     |   call P(A) ;
    |   end ;
    |     |_____niveau 1
end prog ;
    |
    |_____niveau 0
```

La correspondance de ce programme en langage de base est assez simple, quoique fastidieuse, mais d'après les indications données, elle peut être générée automatiquement. Nous le faisons ici en numérotant les unités d'après les niveaux auxquels elles appartiennent.

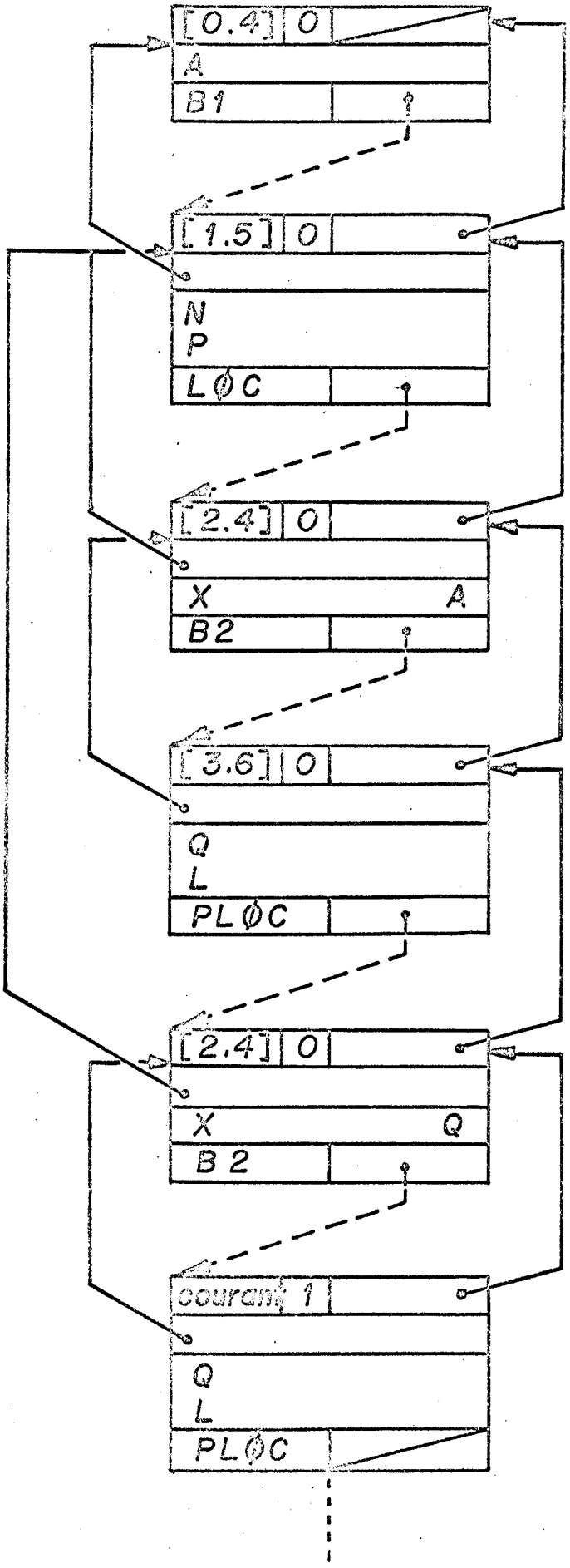
```
    def (PROG, mod (, (I),
[0.1]   (, def (A, taille (1, 'W')), next (*))
[0.2]   (, def (B1, locus), next (*))
[0.3]   (, alloc (B1, act (mod (, (I),
        [1.1]   (, def (N, taille (1, 'W')), next (*))
        [1.2]   (, def (P, mod (,(I),
                [2.1]   (, def (X, fref), next (*))
                [2.2]   (, def (B2, locus), next (*))
                [2.3]   (, alloc (B2, act (mod (, (I),
                        [3.1]   (, def (Q, taille (1, 'W')), next (*))
                        [3.2]   (, def (L, taille (1, 'W')), next (*))
                        [3.3]   (,, cond ((eq (val (L), 0), next (*)),
                                        (true, move (*, E))))
                        [3.4]   (, def (PLOC, locus), next (*))
                        [3.5]   (, alloc (PLOC, act (P, Q)),
                                   ctl (stop (*), next (*), activate
                                   (PLOC • *)))
                        [3.6]   (, desact (PLOC), next (*))
                        [3.7]   (E,, ctl (kill (*), resume (ret (*)))))))),
                         ctl (stop (*), next (*), activate (B2 • *)))
                [2.4]   (, desact (B2), next (*))
                )), next (*))
        [1.3]   (, def (LOC, locus), next (*))
```

```
[1.4]      (, alloc (LOC), act (P, A)), ctl (stop (*), next (*),
                                          activate (LOC • *)))
[1.5]      (, desact (LOC), ctl (kill (*), resume (ret (*)))) ))),
             ctl (stop (*), next (*), activate (B1 • *)))
[0.4]  (, desact (B1), next (*))
       ))
```

Dans le schéma suivant (F18) on a représenté la zone des données des actualisations des modules ; dans chaque zone, la première cellule représente l'index d'exécution du module, dans lequel on a distingué sa valeur (par le numéro de l'unité), son état (1 : prêt, 0 : attente), et l'index de retour par un pointeur. La deuxième cellule est le pointeur vers la zone de données du module englobant. L'exécution se situe après l'appel P(A) et un appel récursif à l'intérieur de P.

Chaînage statique
(portée des variables)

Chaînage dynamique.



Remarques :

(1) - L'allocation des données — donc des variables — des modules se fait suivant un processus de pile (LIFO) ; lorsqu'on revient à un module appelant ou lorsqu'on sort d'un bloc, on désactualise la dernière zone allouée. Cela est une propriété bien connue depuis Algol 60.

(2) - Il y a équivalence entre le chaînage des allocations — en pointillé — qui partent des locus B1, LOC, B2, ... et le chaînage dynamique des index de retour. Cela est dû à ce que, pour les exécutions de blocs, comme pour les appels de procédures traditionnels que l'on reconstitue ici, l'actualisation n'est pas dissociée de l'activation. Il est facile de trouver des exemples où ces chaînages ne sont pas équivalents : plusieurs actualisations simultanées ; plusieurs index de retour, etc... (cf. 6.4.).

6.3.3. Branchements à l'extérieur des blocs

Le problème du branchement à l'extérieur des blocs est délicat puisque la désactualisation est explicitement écrite dans le langage et qu'un module ne peut se désactualiser lui-même, à cause de l'index qui doit être tué. La solution consiste à écrire un module et à l'actualiser dans un contexte d'exécution à structure de blocs, ce module faisant les désactualisations nécessaires à une sortie par branchement.

Exemple : On utilise un formalisme Algol :

```
B1 : begin ;
     ...
     E :
     ...
     B2 : begin ;
          ...
          B3 : begin ;
               goto E ;
          end ;
     end ;
end ;
```

Les instructions équivalentes au goto sont :

(—, ———, ctl (move (B1 • *, E), next (*)))

(, bind (GOTO, (B3, B2)), ctl (kill (*, B2 • *), activate (GOTO • RET : B1 • *)))

Le module actualisé GOTO utilise ses paramètres de la façon suivante :

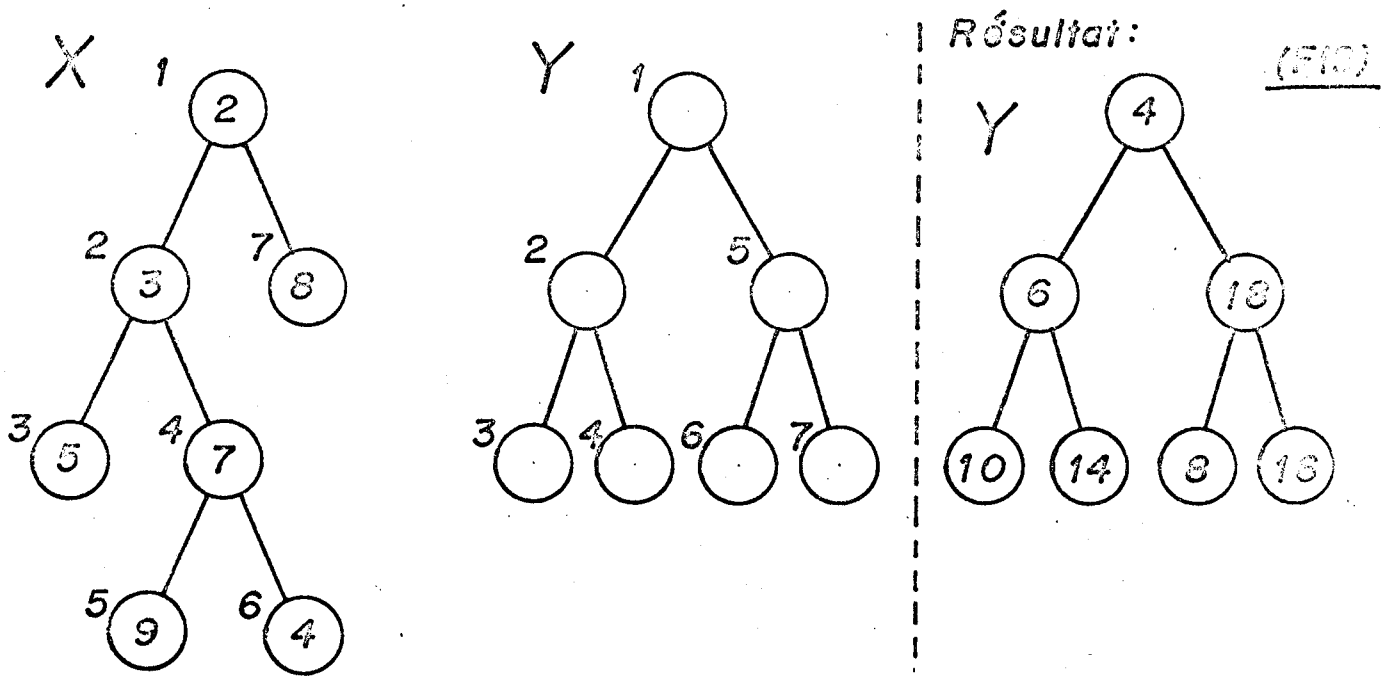
(, desact (B3, B2), ctl (kill (*), resume (ret (*))))

Le retour du module actualisé GOTO se fait donc à l'index B1 positionné à l'étiquette E, et les blocs internes B2 et B3 sont détruits.

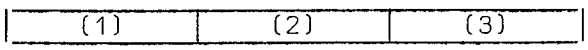
6.4. Exemple d'utilisation de coroutines

Nous reprenons ici un exemple traité en [34] ; il consiste à parcourir en parallèle deux arbres binaires X et Y donnés, ayant le même nombre de noeuds, mais non la même structure. A chaque noeud est associé une valeur entière, et la manipulation a pour but d'affecter à chaque noeud de Y le double de la valeur du noeud correspondant à X.

Le sens du parcours se fait de haut en bas et de gauche à droite, d'où pour le schéma suivant :



L'ensemble est constitué de trois modules : le module principal MP, le module récursif WALK de parcours d'un arbre binaire, le module CALCUL qui fait l'interface entre les deux arbres X et Y. Le module principal actualise deux fois le module WALK, une fois avec l'arbre X en paramètre, une autre fois avec l'arbre Y. La structure d'un noeud de l'arbre est la suivante :



(1) pointeur vers le successeur (ou NIL)

(2) pointeur vers l'alternant (ou NIL)

(3) valeur attachée au noeud.

Un noeud est caractérisé par l'adresse de la structure représentant ce noeud. Les deux premiers éléments de cette structure sont donc des références dont la taille est DEPADR, tandis que le troisième élément a un déplacement d'un mot : dep (1, 'W'). L'arbre lui-même est connu par la donnée de l'adresse de la racine. Voici l'écriture des modules en langage de base, suivis de quelques explications.

Module principal :

```
     def (MP, mod (, (I),
[1]  (, (def (CLOC, locus), def (W1LOC, locus), def (W2LOC, locus)), next (*))
[2]  (, (alloc (W1LOC, act (WALK, fadr)),
         alloc (W2LOC, act (WALK, fadr))), next (*))
[3]  (, alloc (CLOC, act (CALCUL, modif (W1LOC), modif (W2LOC))),
              ctl (activate (W1LOC • *, RET : (CLOC • *, *)),
                   activate (W2LOC • *, ETAT : ATTENTE, RET : (CLOC • *,*)),
                   activate (CLOC • *, ETAT : ATTENTE), stop (*), next
                                                              (*)))
[4]  (MP1, desact (W1LOC), ctl (stop (*), next (*), resume (CLOC • *)))
[5]  (MP2, desact (W2LOC), ctl (kill (CLOC • *), next (*)))
[6]  (, desact (CLOC), kill (*)) ))
```

C'est le module principal qui commence l'exécution :
execute (I, MP, X, Y)

Commentaires.

[1] Définition de trois locus nécessaires aux actualisations.

[2] Actualisation sur W1LOC du module WALK avec le premier formel qui est dans ce cas l'arbre X, et de même sur W2LOC avec le deuxième formel qui est l'arbre Y.

[3] Actualisation du module de calcul sur CLOC avec comme paramètres modifiables, les locus W1LOC et W2LOC, puis activation des index des trois actualisations et passage de contrôle à l'actualisation de W1LOC, les deux autres et le module principal étant mis en attent

[4] Retour de la manipulation : de l'arbre X, il ne reste que l'actual)-sation de la racine, qui est désactualisée, et l'on repart sur le module de calcul qui permet de remonter dans la chaîne des modules de l'arbre Y.

[5] Désactualisation de la racine de Y et fin d'activation du module de calcul.

[6] Désactualisation du module de calcul et fin d'exécution.

Pour une meilleure compréhension, voir le fonctionnement général des modules et leur enchaînement, après les descriptions.

Module de calcul :

Ce module est utilisé en coroutine à partir des deux actualisations de WALK, et de MP.

```
    def (CALCUL, mod (, (I),
[1] (, (def (L1, floc), def (L2, floc)), next (*))
[2] (, use (WALK), ctl (stop (*), next (*), resume (L2 • *)))
[3] (, (def (LOC1, locus), def (LOC2, locus)), next (*))
[4] (, (alloc (LOC1, L1), alloc (LOC2, L2)), move (*, C3))
[5] (C1, alloc (LOC1, L1), ctl (stop (*), next (*), resume (LOC2 • *)))
[6] (C2, alloc (LOC2, L1), next (*))
[7] (C3, store (LOC2 • VAL, mult (2, val (LOC1 • VAL))),
        ctl (move (*, C1), stop (*), resume (LOC1 • *))) ))
```

Commentaires.

[1] Définition de deux locus formels.

[2] La fonction use permet de faire des références externes — codage correct de l'identificateur VAL au [7] — ; arrêt de l'index et activation du module WALK sur la racine de Y.

[3] Retour de la racine de Y. Définition de deux locus.

[4] Affectation des formels aux locus, pour entrer dans la partie de coroutine normale, et branchement à l'unité de calcul [7]. Ces quatre premières unités consituent l'initialisation.

[5] On accède à cette unité à partir d'une actualisation traitant l'arbre X, dont on garde le locus en LOC1, et passage du contrôle à la séquence correspondante pour l'arbre Y.

[6] On accède à cette unité à partir de l'arbre Y ; donc à ce point, on a dans les locus LOC1 et LOC2 les pointeurs vers les actualisations traitant des noeuds correspondant de X et Y.

[7] Séquence de calcul :

valeur du noeud de Y = 2 * valeur du noeud de X ;

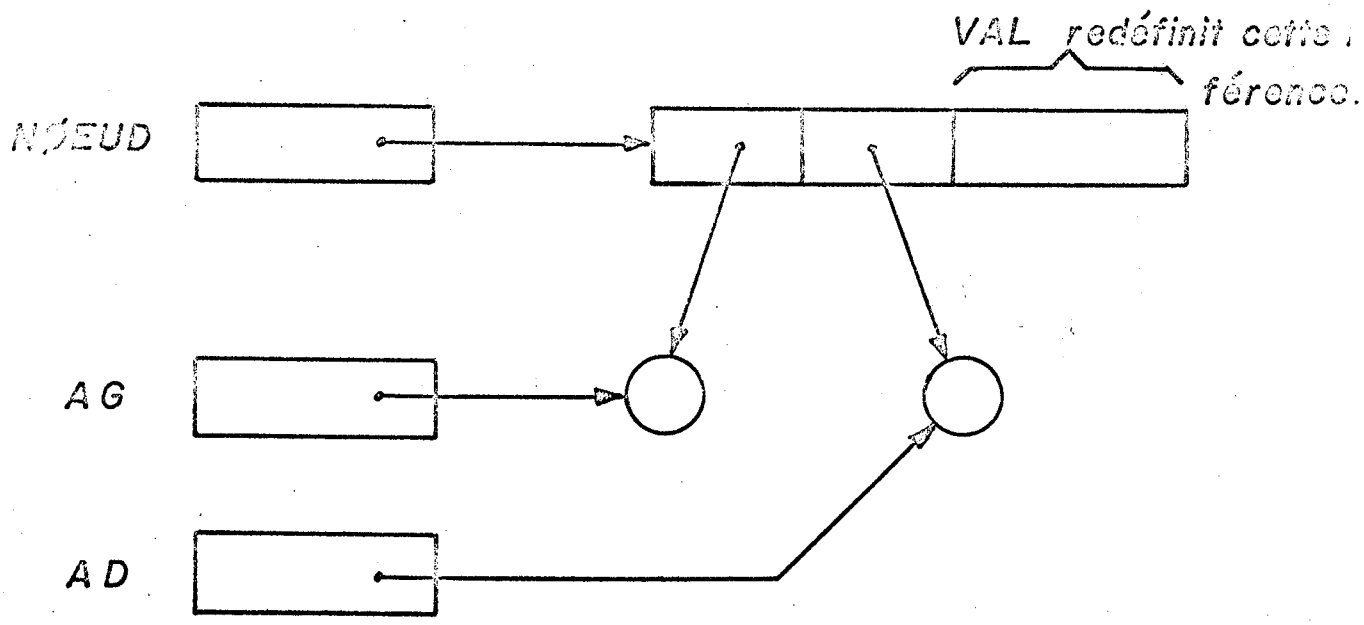
positionnement de l'index en C1 et reprise du traitement de l'arbre X.

Il est à noter que l'index de calcul doit être "tué" dans le module appelant.

Module de parcours d'un arbre binaire :

PRINCIPE : Etant donné un noeud dans un arbre, les variables locales sont positionnées à l'unité [5] de la façon suivante :

(F20)



Les flèches représentent les adresses des éléments sur lesquels elles pointent.

La récursivité de WALK se fait à l'unité [7] sur le sous-arbre gauche ; et à l'unité [9] sur le sous-arbre droit.

```
     def (WALK, mod (, (I),
[1]  (, def (NOEUD, fadr), next (*))
[2]  (, (def (AG, taille (f (DEPADR), u (DEPADR))), def (AD, taille
                          (f (DEPADR), u (DEPADR)))), next (*))
[3]  (, (def (L1, locus), def (L2, locus)), next (*))
[4]  (, (assign (AG, ref (NOEUD, DEPADR)),
         assign (AD, ref (compadr (NOEUD, DEPADR), DEPADR))), next (*))
[5]  (, def (VAL, ref (compadr (NOEUD, addep (DEPADR, DEPADR)), dep (1, 'W'))),
              ctl (stop (*), next (*), resume (ret (*, 1))))
[6]  (W1,, cond ((eq (val (AG), NIL), ctl (kill (*), resume (ret (*, 2)))),
                 (true, next (*))))
[7]  (, (alloc (L1, act (WALK, val (AG))), bind (CALCUL, L1)),
                 ctl (stop (*), next (*), activate (L1 • *, RET : (ret (*, 1),
                                                                   *))))
[8]  (W2, desact (L1), next (*))
[9]  (, (alloc (L2, act (WALK, val (AD))), bind (CALCUL, L2)),
                 ctl (stop (*), next (*), activate (L2 • *, RET : (ret (*, 1),
                                                                   *))))
[10] (W3, desact (L2), ctl (kill (*), resume (ret (*, 2)))) ))
```

Commentaires.

[1] Définition de l'adresse formelle NOEUD.

[2] Références locales contenant les adresses des noeuds "fils".

[3] Locus des actualisations relatives aux deux branches.

[4] Affectation des valeurs d'adresses à AG et AD.

[5] Définition de la référence VAL et passage au pas [5] ou [6] de CALCUL.

[6] Test sur la feuille de l'arbre ; dans le cas où il n'y a pas de fils, on revient au module appelant (au noeud "père").

[7] Actualisation de WALK pour traiter la branche gauche ; modificatio du paramètre de calcul et activation de l'actualisation.

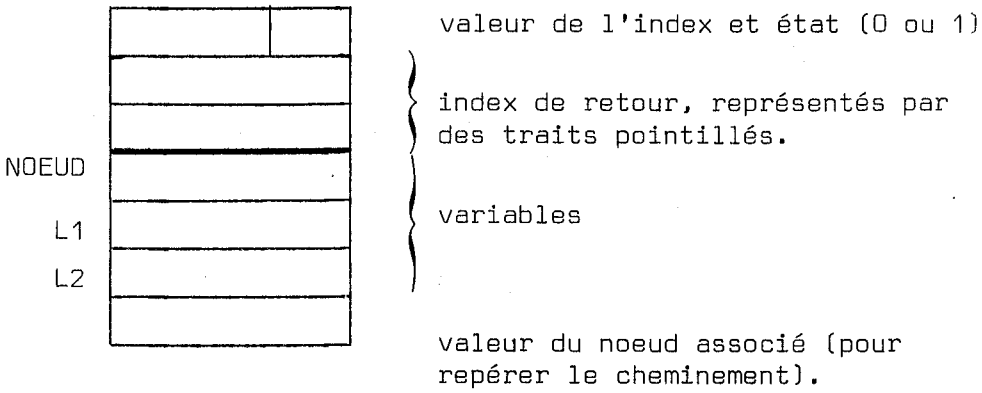
[8] Retour d'une branche gauche ; désactualisation du module d'où l'o vient.

[9] Action identique à [7] sur la branche droite.

[10] Retour d'une branche droite ; fin du module ; désactualisation du module d'où l'on vient ; retour au module du noeud "père".

SCHEMAS DONNANT L'EVOLUTION DES DONNEES AU COURS DE L'EXECUTION :

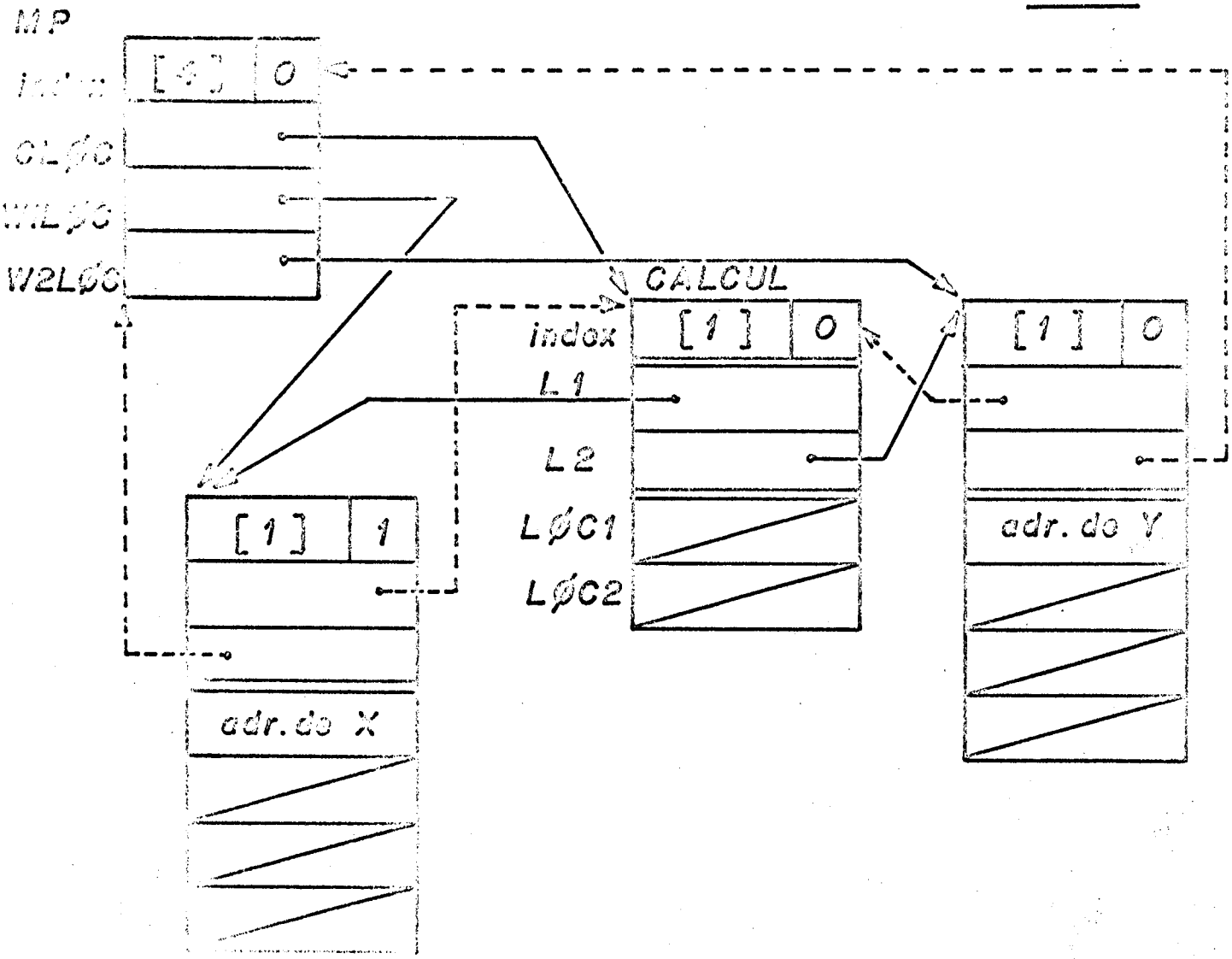
On représente la zone de données d'un module WALK actualisé pa



Les locus sont symbolisés par une flèche pointant sur la zone de données ; on ne représente pas la partie du locus qui repère le code à exécuter, car le code n'est présent qu'une fois.

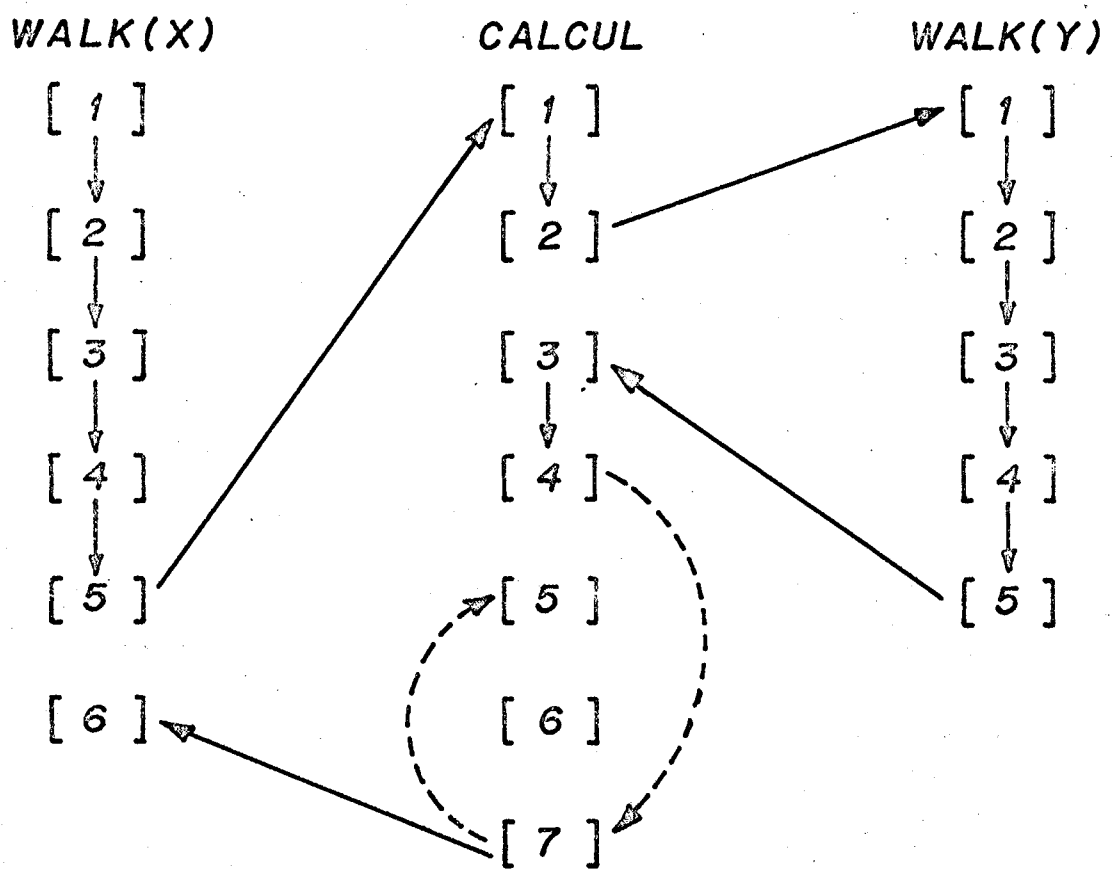
Après actualisation par le module principal, on a :

(F21)



Le passage de contrôle entre les différentes unités pour les initialisations est représenté par le diagramme suivant, à partir de l'état donné dans la figure F21.

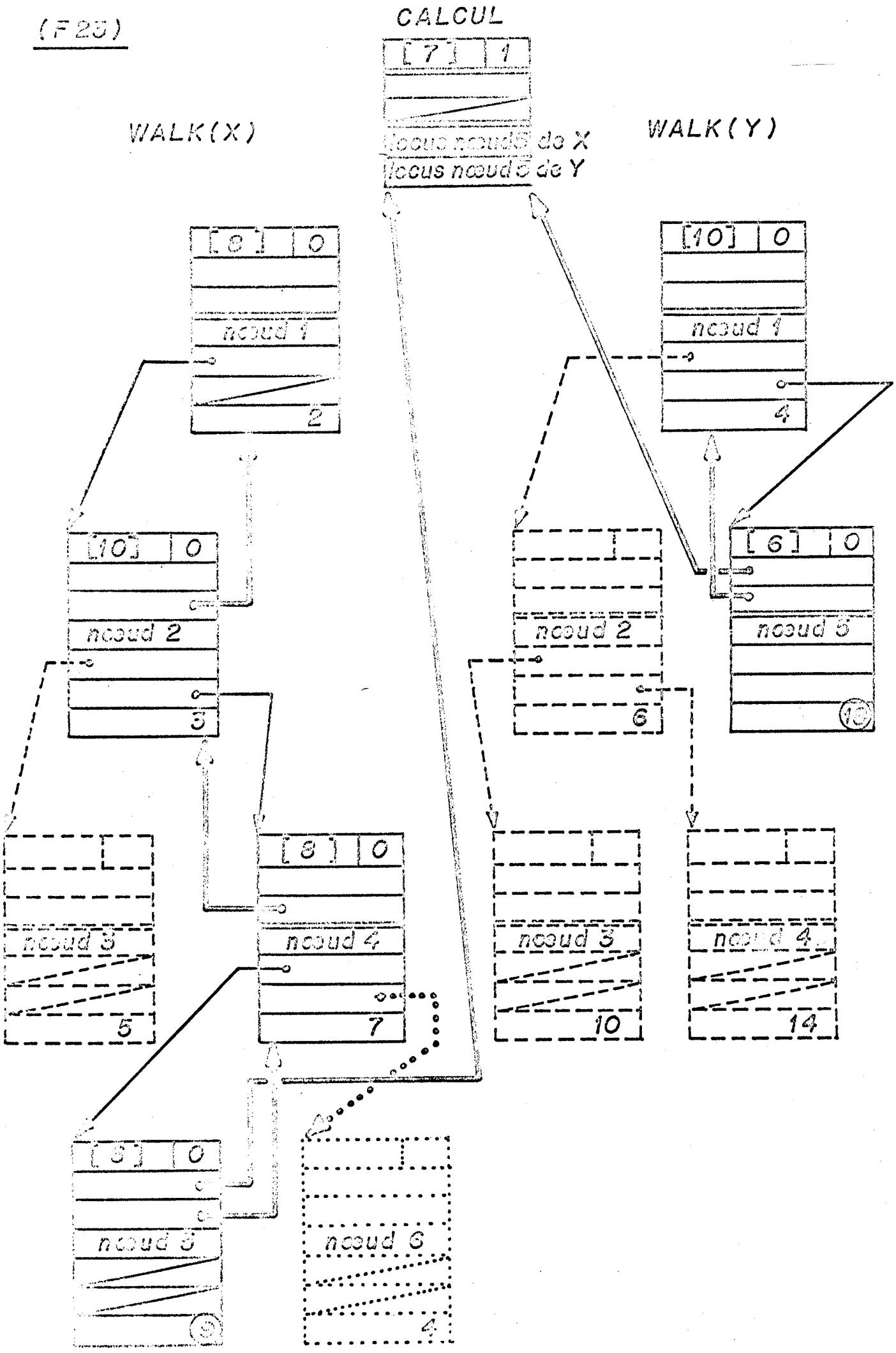
(F22)

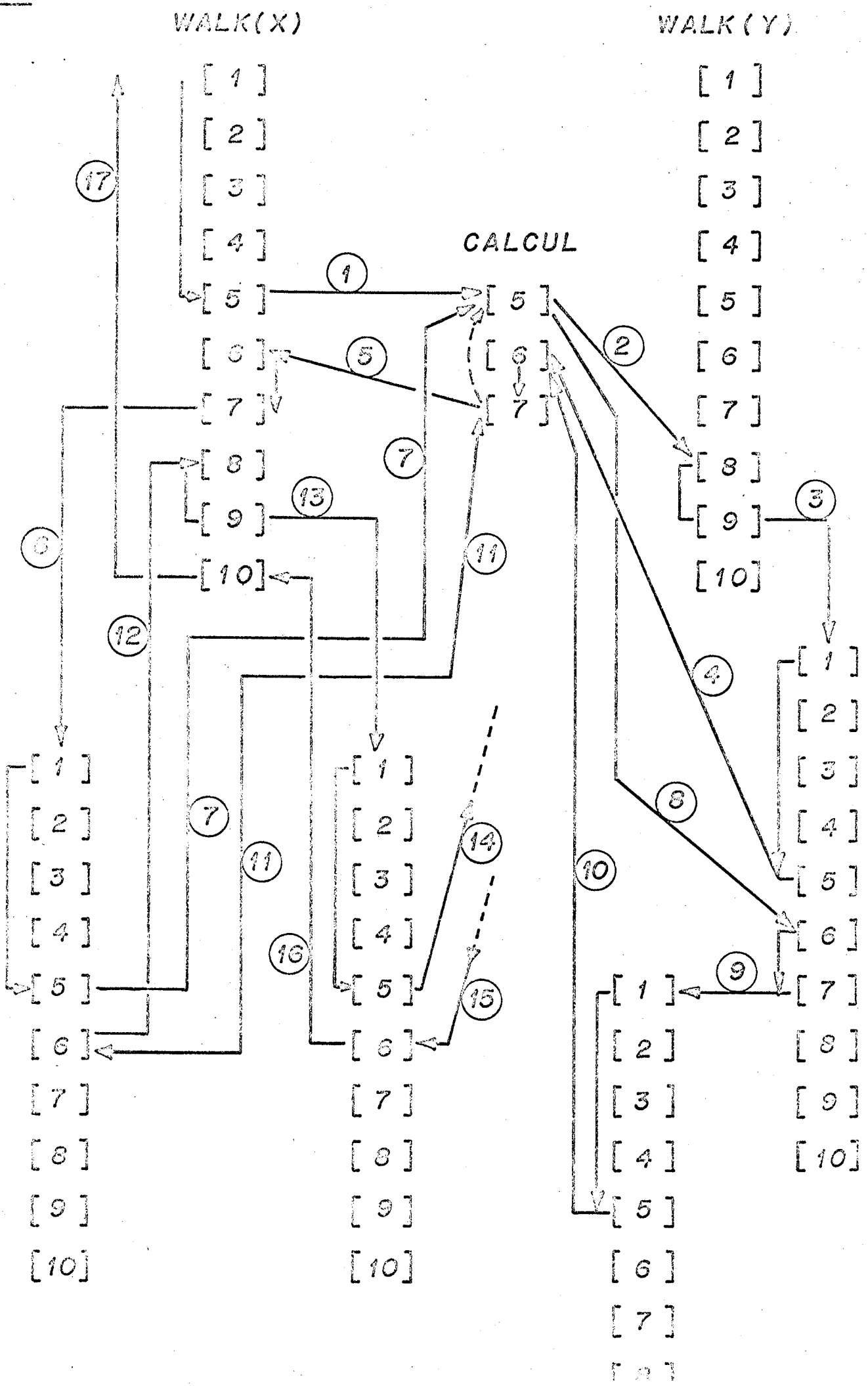


La flèche pointillée indique le repositionnement de l'index du module de calcul.

Les actualisations du module WALK suivent le cheminement dans les arbres. La liaison des données au niveau de récursivité qui correspond au noeud 5 des arbres X et Y est schématisé dans la figure F23. Les modules en tirets ont été désactualisés. Le premier index de retour est toujours celui du module de calcul ; les autres sont symbolisés par une double flèche. On se place sur la figure au moment où le module de calcul est actif et calcule la valeur (18) à partir de (9) . Dans l'étape suivante, le contrôle passe au module du noeud 5 de l'arbre X ; celui-ci revient alors en 4 et il y a actualisation du noeud 6 (en pointillé), alors que le noeud 5 est désactualisé.

(F 23)


CALCUL
[7] 1
locus noeud 5 de X
locus noeud 5 de Y
WALK(X)
WALK(Y)
[8] 0
noeud 1
2
[10] 0
noeud 1
4
[10] 0
noeud 2
3
noeud 2
6
[6] 0
noeud 5
12
noeud 3
5
[8] 0
noeud 4
7
noeud 3
10
noeud 4
14
[3] 0
noeud 3
9
noeud 6
4


WALK(X)
WALK(Y)
CALCUL
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

Le passage dynamique du flot de contrôle est détaillé à la figure F24, suivant les divers cas de configuration des noeuds. On a représenté les actualisations en recopiant les indices des unités, afin de ne pas trop embrouiller le schéma, mais le code du module WALK n'est présent qu'à un seul exemplaire dans tout le déroulement du programme. La flèche indique le cheminement du flot, dans l'ordre des numéros encerclés. On reconnaît ② ③ l'actualisation et le passage de contrôle à un sous-arbre droit ; ⑧ ⑨ , le même processus pour un sous-arbre gauche. Les va-et-vient à travers le module de CALCUL sont marqués par ① ② , ④ ⑤, ⑦ ⑧ , ⑩ ⑪ .

Les retours au noeud "père" sont symbolisés par ⑫ : retour d'une feuille gauche ; ⑯ : retour d'une feuille droite ; ⑰ : retour d'un noeud intermédiaire.

6.5. Dialecte de programmes non déterministes

On traite ici l'exemple complet de transformation d'algorithme non déterministe en algorithme déterministe donné par Floyd [16]. Le principe de la "marche arrière" a été donné en 3.4.10 ; cette partie en est l'application.

6.5.1. Définition et utilisation des piles

Pour définir et utiliser les piles, on souhaite disposer de trois objets :

(1) définition : def (P, pile (1, 'W'))

où les paramètres de pile représentent la place occupée par un élément à mettre dans la pile.

(2) empiler : empiler (P, a)

opérateur qui met la valeur "a" au sommet de la pile P.

(3) dépiler : depiler (P)

opérateur qui donne la valeur du sommet de pile et l'enlève de ce sommet.

La déclaration de la pile peut être traitée par une simple macro syntaxique faisant le remplacement :

```
<unite> → (<etiq>, <defexpr>, <contrôle>) =

  [0]    (<defexpr> → def (<identificateur>, pile (<f>, <u>)) |

  [1]    (<etiq>, def (DEPPILE, addep (DEPADR, addep (DEPDEP,
                  dep (mult (PROFONDEUR, <f>), <u>)))), next (*))

  [2]    (, def (<identificateur>, taille (f (DEPPILE), u (DEPPILE))),
                                                  next (*))

  [3]    (, store (ref (orig (<identificateur>), DEPADR),
                       compadr (orig (<identificateur>), addep
                                   (DEPADR, DEPDEP))), next (*))

  [4]    (, store (ref (compadr (orig (P), DEPADR), DEPDEP), dep (<f>, <u>))
                                                  contrôle>) | = ) ;
```

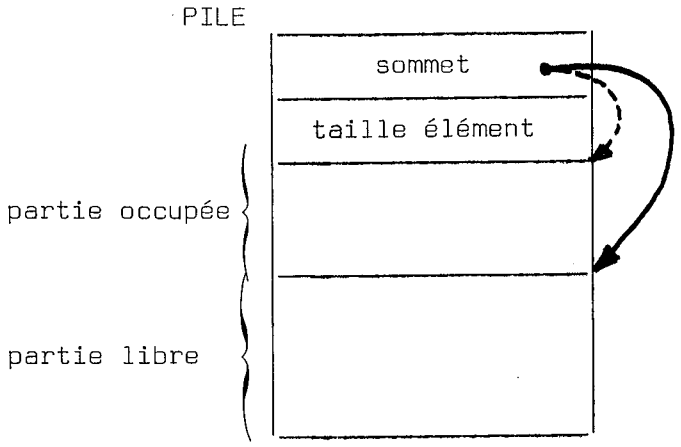
Explication de la macro :

L'entête [0] indique que lorsqu'on trouve une définition de pile dans une unité, on fait le remplacement par les unités [1, 2, 3, 4] sinon on fait une analyse normale (sens du symbole "=") [8].

L'unité [1] calcule la taille prise par la pile ; cette taille peut être calculée directement par le macro-processeur si celui-ci effectue des opérations, sinon il y a génération de l'identificateur DEPPILE — comme toutes les générations d'identificateurs, il faut prévoir le mécanisme évitant les doubles définitions — et le calcul de la taille est fait à l'exécution, calcul dépendant de PROFONDEUR qui est une constante donnant la profondeur de la pile.

L'unité [2] définie la variable de type pile que l'on veut utiliser ; c'est simplement une "référence". Les unités [3] et [4] initialisent les deux premiers éléments de la pile qui seront utilisés par les opérateurs, à savoir, la position du sommet pile (une adresse) et la taille d'un élément (un déplacement).

La pile en état d'utilisation est schématisée par :



- en pointillés, la position initiale du pointeur.

Définition des opérateurs.

On définit les opérateurs à partir de modules :

```
def (EMPMOD, mod (, (I),
    (, def (PILE, fref), cond ((eqadr (val (ref (orig (PILE), DEPADR)),
                                       compadr (orig (PILE), tref (PILE))),
                                  move (*, L)),
                                 (true, next (*))))
    (, def (ELEM, ref (val (ref (orig (PILE), DEPADR)),
                       val (ref (compadr (orig (PILE), DEPADR), DEPDEP)))),
                                                        next (*))
    (, store (ref (orig (PILE), DEPADR), compadr (orig (ELEM), tref (ELEM))),
                       ctl (kill (*), resume (ret (*))))
    (L, message ('DEPASSEMENT CAPACITE PILE'), ctl (kill (*), resume (ret
                                                        (*)))) ))

def (EMPOP, opmod (EMPMOD,I, stack (ELEM)))

def (EMPILER, opcomp (store (empop (fref), fconst)))
```

Le module EMPMOD produit les actions suivantes :

- test sur la fin de pile
- définition de la référence (ELEM) donnant le sommet de pile
- mise à jour de l'adresse du sommet .

L'opérateur EMPOP compile le module et donne comme résultat la référence au sommet de pile ; l'opérateur EMPILER est celui que l'on veut utiliser.

```
def (DEPMOD, mod (, (I),
    (, def (PILE, fref), cond (( eqadr (val (ref (orig (PILE), DEPADR)),
                                  compadr (orig (PILE), addep (DEPADR, DEPDEP))),
                                move (*, L)),
                               ( true, next (*))))
    (, def (ELEM, ref (sousadr (val (ref (orig (PILE), DEPADR)),
                                val (ref (compadr (orig (PILE), DEPADR),
                                                  DEPDEP))),
                            val (ref (compadr (orig (PILE), DEPADR), DEPDEP)))),
                                                  next (*))
    (, store (ref (orig (PILE), DEPADR), orig (ELEM)),
                        ctl (kill (*), resume (ret (*))))
    (L, message ('PILE VIDE'), ctl (kill (*), resume (ret (*)))) ))
def (DEPOP, opmod (DEPMOD, I, stack (ELEM)))
def (DEPILER, opcomp (val (depop (fref))))
```

Les définitions qui conduisent à l'opérateur DEPILER sont sur le même modèle que celles de EMPILER.

6.5.2. Transformation d'algorithme non déterministe en algorithme déterministe.

La transformation est donnée sur un graphe par Floyd ; on la donne ici sur les unités, mais le principal problème est celui de résoudre les branchements. Pour cela, un préprocesseur doit traiter les étiquettes et les "move" (les next sont assimilés aux move). Pour chaque unité du programme non déterministe :

(etiquette, expression, contrôle)

correspond au moins deux unités du programme déterministe :

(etiquette, développement expression, contrôle*)

(étiquetteprim, operation de retour, contrôleprim)

Le "contrôleprim" est généré d'après certaines informations sur le graphe de branchement du programme, de même que le "contrôle*" est issu du "contrôle" avec parfois certaines modifications.

(1) - TRAITEMENT DES BRANCHEMENTS -

On suppose que toutes les unités sont étiquetées. Le graphe des branchements est représenté par un tableau. Soient les unités :

(ETIQ1, ..., cond ((p, move (*, ETIQ2)), (true, move (*, ETIQ3))))
.
.
.
(ETIQA, ..., move (*, ETIQ1))
.
.
.
(ETIQB, ..., move (*, ETIQ1))
.
.
.
(ETIQC, ..., move (*, ETIQ1))
.
.
.
(ETIQ2, ...)
(ETIQ3, ...)
(ETIQM-1, ..., move (*, ETIQM))
(ETIQM, ..., move (*, ETIQM+1))
(ETIQM+1, ...

} mis pour next (*).

Le tableau des branchements est, pour ETIQ1 et ETIQM :

Etiquette unité	Etiquettes successeurs	Etiquettes prédécesseurs
ETIQ1	ETIQ2, ETIQ3	ETIQA, ETIQB, ETIQC
ETIQM	ETIQM+1	ETIQM-1

A partir de ce tableau, les instructions de contrôle générées dépendent du nombre d'étiquettes prédécesseurs ou successeurs. Pour l'exemple donné, on donne une génération à deux index, un index pour le

déroulement normal : I, un index pour le retour : IPRIM. M est une pile, TEST une variable.

```
(ETIQ1, ..., cond ((p, move (I, ETIQ2)), true, move (I, ETIQ3))))
(ETIQ1PRIM, store (TEST, depiler (M)),
                        cond ((eq (val (TEST), 0), move (IPRIM, ETIQAPRIM)),
     .                        (eq (val (TEST), 1), move (IPRIM, ETIQBPRIM)),
     .                        (eq (val (TEST), 2), move (IPRIM, ETIQCPRIM)))
(ETIQA, (..., empiler (M,0)), move (I, ETIQ1))
(ETIQAPRIM, ...
     .
     .
     .
(ETIQB, (..., empiler (M,1)), move (I, ETIQ1))
(ETIQBPRIM, ...
     .
     .
     .
(ETIQC, (..., empiler (M,2)), move (I, ETIQ1))
(ETIQCPRIM, ...
     .
     .
     .
(ETIQ2, ...
(ETIQ2PRIM, ..., move (IPRIM, ETIQ1PRIM))
     .
     .
     .
(ETIQ3, ...
(ETIQ3PRIM, ..., move (IPRIM, ETIQ1PRIM))

(ETIQM, ..., move (I, ETIQM+1))
(ETIQMPRIM, ..., move (IPRIM, ETIQM-1PRIM))
```

(2) - TRAITEMENT DES OPERATIONS -

Certaines opérations sont spécifiques de la "marche arrière" : l'ECHEC, le SUCCES, la procédure CHOIX ; d'autres doivent être traitées d'une manière particulière, ce sont : l'affectation, l'appel de sous-programmes, l'entrée et la sortie des sous-programmes, la lecture et l'écriture des données.

Pour traiter ces opérations, on dispose de trois piles, une pour les variables : M, une pour les entrées : R, une pour les sorties : W. On suppose pour simplifier que les variables ont le même format ; si ce n'est pas le cas, il faut une pile pour chaque type de variables.

Transformation des opérations impliquées dans la "marche arrière" :

On donne l'écriture d'une unité pour le programme non déterministe et son équivalent dans le programme déterministe correspondant.

(1) - (E, ———, echec) =>

$$\begin{cases} (E, \text{———}, \underline{\text{ctl}}\ (\underline{\text{stop}}\ (I), \underline{\text{move}}\ (IPRIM, EPRIM), \\ \qquad\qquad\qquad\qquad\qquad\qquad \underline{\text{resume}}\ (IPRIM)) \\ (EPRIM,, cprim) \end{cases}$$

echec est considéré comme une fonction de contrôle. Le contrôle "cprim" est calculé en fonction des unités "prédécesseur" de l'unité E comme il a été indiqué pour le traitement des branchements.

(2) - (E, ———, succès) =>

$$\begin{cases} (E, (\text{———}, \underline{\text{imprimer}}\ (* \cdot BACK \cdot W)), \underline{\text{ctl}}\ (\text{stop}\ (I), \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{move}\ (IPRIM, EPRIM), \\ \qquad\qquad\qquad\qquad\qquad\qquad \underline{\text{resume}}\ (IPRIM)) \\ (EPRIM,, cprim) \end{cases}$$

De la même manière que l'"echec", le "succès" amène un changement d'index et un retour arrière dans le programme, après impression de la pile du résultat. Dans le cas où l'algorithme n'exige pas le retour complet, on peut arrêter et même tuer les deux index après le "succès". La notation * • BACK • W pour désigner la pile W est conditionnée par l'utilisation faite en 6.5.3. On pourrait généraliser par l'utilisation de formels.

(3) - (E, choix (X, n), c) ⇒

(E, empiler (⋆ • BACK • M, val (X)), next (I))

(♯E1, store (X, n), next (I))

(♯E2,, cond ((sup (val (X), 0), c),
(true, next (I))))

(, store (X, depiler (⋆ • BACK • M)), ctl (stop (I), resume (IPRIM),
cprim))

(EPRIM, store (X, moins (val (X), 1)), ctl (stop (IPRIM), resume (I),
move (I, ♯E2)))

Les étiquettes marquées d'un '♯' sont des étiquettes générées — avec les conditions de génération pour éviter les doubles définitions —. Il y a ici plus de deux unités, mais on voit l'emplacement du contrôle 'c' et celui de 'cprim' ; EPRIM est l'étiquette retour de l'unité E.

(4) - Affectation : (E, storeb (X, f), c) ⇒

(E, empiler (⋆ • BACK • M, val (X)), next (I))

(, store (X, f), c)

(EPRIM, store (X, depiler (⋆ • BACK • M)), cprim)

(5) - Appel de sous-programmes non déterministes. Pour pouvoir générer automatiquement le sous-programme déterministe correspondant, on suppose que le sous-programme n'a qu'une entrée et une sortie, ce qu'il est toujours possible de faire en ajoutant des unités de branchement. Le sous-programme est un module à un index qui est transformé en module à deux index.

(E, call (A), c) ⇒

(E, alloc (♯LA, act (A)), ctl (move (I, ♯E1), move (IPRIM, ♯E1PRIM)))

(♯E1,, ctl (activate (♯LA • I), stop (I), move (I, ♯E2)
activate (♯LA • IPRIM, etat : ATTENTE, ret : IPRIM),
move (IPRIM, ♯E2)))

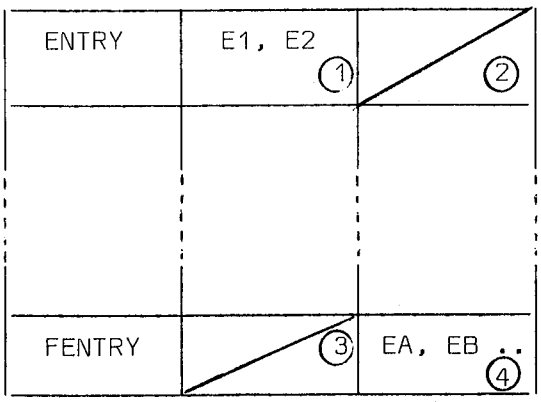
(♯ E1PRIM,, ctl (activate (♯ LA • I, etat : ATTENTE, ret : I),
move (I, ♯ E2),
activate (♯ LA • IPRIM), stop (IPRIM), next (IPRIM)))

(♯ E2, desact (♯LA), ctl (c, cprim))

(EPRIM,, move (IPRIM, E))

(6) - Ecriture des sous-programmes.

Le tableau des étiquettes doit se présenter ainsi :




```
A : beginb (ANALYSEUR)
    (ENTRY, ...)
       .<suite d'unités du module>
       .
       .
     endb
```

L'expansion est :

```
def (A, mod ((I, IPRIM),
    (, use (BACKTRACKING, ANALYSEUR), ctl (move (I, ENTRY)
                                           move (IPRIM, FENTRYPRIM)))
    (ENTRY, ..., c)
    (ENTRYPRIM, ..., ctl (kill (I, IPRIM), resume (ret (IPRIM))))
         .
         .<suite des unités>
         .
    (FENTRY, ..., ctl (kill (I, IPRIM), resume (ret (I))))
    (FENTRYPRIM,, cprim)
    ))
```

L'étiquette du beginb devient l'identificateur du module ; les identificateurs entre parenthèse (ici, ANALYSEUR) sont introduits dans la liste du use. Cette insertion est nécessaire, non seulement pour les variables du retour arrière (piles, opérateurs, ...) mais encore pour les variables utilisés dans le sous-programme qui doivent être définies dans le module appelant initial, à l'exception des variables générées comme les locus des actualisations de modules (exemple, ↓LA dans l'expansion du call (A)). Ces définitions sont insérées dans l'unité du use. Dans le tableau des étiquettes, les cases (2) et (3) sont vides ; elles correspondent respectivement aux parties contrôle des unités ENTRYPRIM (retour par IPRIM) et FENTRY (retour par I). La case (1) donne le contrôle noté 'C' dans l'expansion, tandis que la case (4) donne le contrôle noté 'cprim'. Il y a une parfaite dualité entre début-fin de module et contrôles de I et IPRIM.

(7) - Impression des résultats :

(E, print (X), c) ⇒ { (E, empiler (* • BACK • W, X), c)
                    { (EPRIM, depiler (* • BACK • W), cprim)

Pour la lecture des données, on ne peut faire des retours arrière dans le fichier d'entrée, en général séquentiel, d'où une lecture physique faite par le module initial dans le buffer R définit dans le module de "backtracking". Une référence CC contient le caractère courant de lecture logique ; deux opérateurs, avancer et reculer font déplacer la tête de lecture logique.

(E, lireb ('a'), c) ⇒

(E,, cond ((eq (val (* • BACK • CC), 'a'), next (I)),

(true, ctl (stop (I), resume (IPRIM), cprim))))

(, avancer (* • BACK • R), c)

(EPRIM, reculer (* • BACK • R), cprim)

Cette expansion lireb correspond à la comparaison d'un caractère terminal pour l'algorithme d'analyse qui sera traité au paragraphe suivant. Le test de l'unité E vérifie si le caractère courant est bien celui attendu

('a') ; si c'est le cas, on avance la tête de lecture et l'on passe à la suite, sinon, on génère un "echec", ce qui fait revenir en arrière. Lorsqu'on arrive en EPRIM, c'est que l'on est tombé sur une lecture infructueuse plus loin ; il faut donc annuler la lecture faite ici en reculant dans la chaîne d'entrée.

6.5.3. Application à l'analyse syntaxique

Un exemple d'application de la méthode de "marche arrière" peut être l'analyse syntaxique. On a choisi un analyseur descendant non déterministe. Les éléments donnés précédemment permettent d'écrire facilement cet analyseur.

Les modules initiaux sont :

Le module BACKTRACKING qui contient les définitions de :

- M, W : piles ;
- TEST : référence contenant un entier ;
- R : zone de lecture ;
- CC : référence contenant un caractère ;
- les opérateurs : empiler, depiler, avancer, reculer, imprimer.

Le module qui actualise BACKTRACKING et lance l'analyse :

```
def (ANALYSEUR, mod (,(I, IPRIM),
    (, (def (BACK, locus), def (AXIOME, locus), use (BACKTRACKING)), next (I))
    (, alloc (BACK, act (BACKTRACKING)), ctl (stop (I), next (I),
                                              activate (BACK • *)))
    (, store (BACK • R, fconst), ctl (activate (IPRIM, etat : ATTENTE),
                                      next (I)))
    (, def (N, taille (1, 'W')), next (I))
    (, alloc (AXIOME, act (S)), ctl (stop (I), next (I), move (IPRIM, L2),
                                     activate (AXIOME • I),
                                     activate (AXIOME • IPRIM, etat : ATTENTE,
                                               ret : IPRIM)))
```

(L1, imprimer ('CHAINE CORRECTE'), move (I, L3))

(L2, imprimer ('ERREUR SYNTAXIQUE'), next (IPRIM))

(L3, desact (AXIOME, BACK), ctl (kill (I, IPRIM), resume (ret (I))))))

Dans le module ANALYSEUR, fconst est la chaîne d'entrée, S est le module correspondant à l'axiome de la grammaire (il pourrait aussi être donné en paramètre). L'exemple de grammaire choisi est :

G : S → |— A —|

A → aAB | a

B → aB | b

qui génère les chaînes du type :

$$|— a^n\ a\ (a^*b)^n —|$$ avec $n \geqslant 0$

A partir de la grammaire G et des primitives définies dans le paragraphe précédent, l'écriture des modules est immédiate :

```
S : beginb (ANALYSEUR)
    (S1, lireb ('|—'), next (I))
    (S2, call (A), next (I))
    (S3, lireb ('—|'),)
endb

A : beginb (ANALYSEUR)
    (A1, choix (* • N,2), cond ((eq (val (* • N), 1), move (I, A2)),
                                (eq (val (* • N), 2), move (I, A5))))
    (A2, lireb ('a'), next (I))
    (A3, call (A), next (I))
    (A4, call (B), move (I, FA))
    (A5, lireb ('a'), next (I))
    (FA,,)
endb
```

```
B : beginb (ANALYSEUR)
    (B1, choix (* • N, 2), cond ((eq (val (* • N), 1), move (I, B2)),
                                 (eq (val (* • N), 2), move (I, B4))))
    (B2, lireb ('a'), next (I))
    (B3, call (B), move (I, FB))
    (B4, lireb ('b'), next (I))
    (FB,,)
endb
```

Le symbole * en position de locus (* • N) signifie qu'il s'agit du locus du système utilisé lors du execute. On donne, en effet, une chaîne à analyser par :

```
execute (I, ANALYSEUR, '⊢aab⊣')
```

D'après la syntaxe abstraite (5.4), il y a **actualisation** du module ANALYSEUR sur un locus du système et activation de l'index I de cette actualisation. Pour accéder dynamiquement aux variables de cette actualisation, on utilise la notation * pour le locus anonyme du système. Les étiquettes FA et FB sont nécessaires pour n'avoir qu'une seule sortie dans le module. L'index spécifié pourrait être * à la place de I, puisque l'index courant dans le sens d'exécution en avant est toujours I (IPRIM étant réservé pour le retour).

A titre d'exemple, on donne la transformation du module B en module déterministe.

Tableau des étiquettes :

	Success.	Prédéc.
B1	B2, B4	-
B2	B3	B1
B3	FB	B2
B4	FB	B1
FB	-	B3, B4

```
def (B, mod (,(I, IPRIM),
    (, (use (BACKTRACKING, ANALYSEUR),
        def (#LB, locus)), ctl (move (I, B1), move (IPRIM, FBPRIM)))
    (B1, empiler (* • BACK • M, val (* • N)), next (I))
    (#B11, store (* • N, 2), next (I))
    (#B12,, cond (( sup (val (* • N), 0), cond ((eq (val (* • N, 1),
                                                    move (I, A2)),
                                                (eq (val (* • N, 2),
                                                    move (I, A5)))),
                                    (true, next (I))))
    (, store (* • N, depiler (* • BACK • M)), ctl (stop (I), resume (IPRIM)
                                                  kill (I, IPRIM), resume
                                                      (ret (IPRIM))))
    (B1PRIM, store (* • N, minus (val (* • N), 1)),
            ctl (stop (IPRIM), resume (I), move (I, #B12)))
    (B2,, cond ((eq (val (* • BACK • CC), 'a'), next (I)),
                (true, ctl (stop (I), resume (IPRIM), move (IPRIM, B1PRIM)))))
    (, avancer (* • BACK • R), move (I, B3))
    (B2PRIM, reculer (* • BACK • R), move (IPRIM, B1PRIM))
    (B3, alloc (#LB, act (B)), ctl (move (I, #B31), move (IPRIM, #B31PRIM)))
    (#B31, empiler (M, 0), ctl (activate (#LB • I), stop (I), move (I, #B32),
                activate (#LB • IPRIM, etat : ATTENTE, ret : IPRIM),
                move (IPRIM, #B32)))
    (#B31PRIM,, ctl (activate (#LB • I, etat : ATTENTE, ret : I), move
                                                        (I, #B32),
                activate (#LB • IPRIM), stop (IPRIM), next (IPRIM)))
    (#B32, desact (#LB), ctl (move (I, FB), move (IPRIM, B2PRIM)))
    (B3PRIM,, move (IPRIM, B3))
    (B4, empiler (M, 1), cond ((eq (val (* • BACK • CC), 'b'), next (I)),
                (true, ctl (stop (I), resume (IPRIM), move (IPRIM, B1PRIM))))
    (, avancer (* • BACK • R), move (I, FB))
```

```
(B4PRIM, reculer (* • BACK • R), move (IPRIM, B1PRIM))

(FB,, ctl (kill (I, IPRIM), resume (ret (I))))

(FBPRIM, store (* • BACK • TEST, depiler (M)),
                    cond ((eq (val (TEST), 0), move (IPRIM, B3PRIM)),
                          (eq (val (TEST), 1), move (IPRIM, B4PRIM)))) ))
```

Remarques :

(1) La transformation de l'algorithme peut être faite automatiquement, mais on constate qu'une analyse avec retour arrière sera beaucoup moins efficace qu'avec une méthode plus élaborée (exemple [14]).

(2) La marche arrière peut être faite avec un seul index, puisqu'il n'y en a qu'un seul actif à un moment donné, mais on est alors obligé d'introduire un test supplémentaire au retour des sous-programmes.

(3) Cette méthode descendante ne peut pas s'appliquer pour des règles récursives à gauche.

6.6. Langage classifié

On désire utiliser le mécanisme des classes (cf. Ch.4) pour générer les conversions et employer les procédures génériques dans un dialecte de type Algol. Pour cela, il faut donner :

- les définitions des classes, des conversions et des procédures ;
- un ensemble d'extensions syntaxiques permettant à l'utilisateur d'écrire de façon lisible.

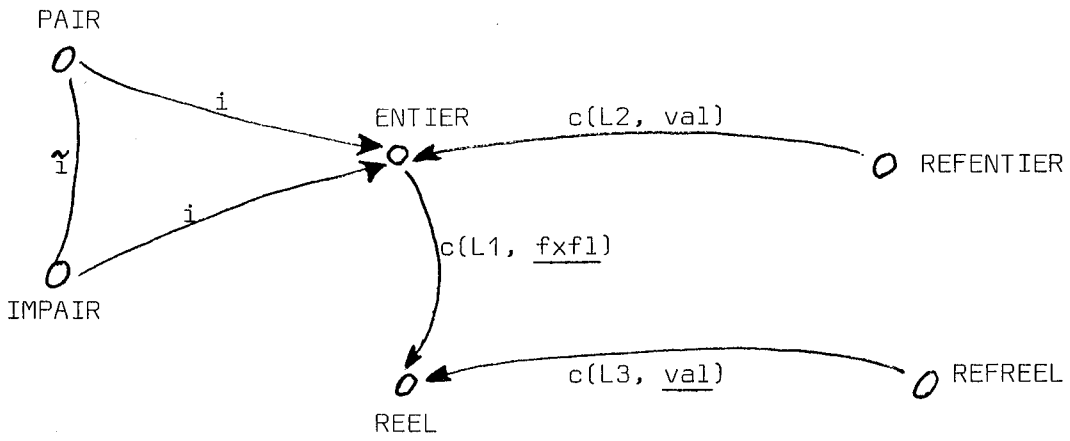
6.6.1. Définitions des classes du dialecte

On s'intéresse à une application orientée vers l'analyse numérique, d'où l'ensemble des définitions où apparaissent les classes ENTIER, REEL, etc... ainsi que les procédures des opérations arithmétiques usuelles. Les opérateurs utilisés sont ceux du langage de base. (cf. Annexe 1).

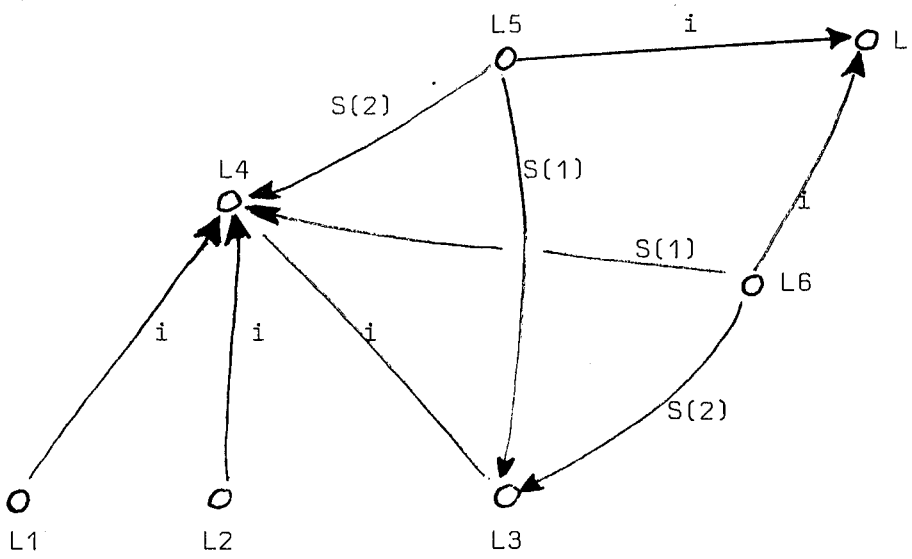
```
declass (
    def (ENTIER, class), def (REEL, class),
    def (REFENTIER, class), def (REFREEL, class),
    def (PAIR, in ENTIER), def (IMPAIR, compl (PAIR, ENTIER)),
    def (VIDE, class),
    def (L1, in L4), def (L2, in L4), def (L3, in L4),
    def (L4, level),
    def (L5, cart (L3, L4), def (L6, cart (L4, L3)),
    def (L, union (L5, L6))
    def (VAL, conv (L2, REFENTIER, ENTIER, val (fref))),
    def (VAL, conv (L3, REFREEL, REEL, val (fref))),
    def (, conv (L1, ENTIER, REEL, fxfl (fconst))),
    def (ASSIGN, proc (cart (L1, L2), cart (REFENTIER, ENTIER),
                              VIDE, store (fref, fconst))),
    def (ASSIGN, proc (cart (L1, L4), cart (REFREEL, REEL), VIDE,
                                    store (fref, fconst))),
    def (PLUS, proc (cart (L2, L2), cart (ENTIER, ENTIER), ENTIER,
                                          plus (fconst, fconst))),
    def (PLUS, proc (L, cart (REEL, REEL), REEL,
                                          plusfl (fconst, fconst))),
    def (ABS, proc (L2, ENTIER, ENTIER, abs (fconst))),
    def (ABS, proc (L3, REEL, REEL, absfl (fconst))) )
```

Sur le même modèle que PLUS, on pourrait écrire les procédures génériques de MOINS, MULT, DIV, ...

6.6.2. Graphe des classes



6.6.3. Graphe des niveaux



6.6.4. Remarques

(1) - L'introduction des niveaux permet de lever l'ambiguité, comme le montre l'exemple de ABS. Lorsqu'on a :

apply (ABS, X)

si X est ENTIER ou REEL, le problème est simple puisqu'il n'y a pas de conversions à appliquer. Si par contre X est un REFENTIER, il pourrait y avoir confusion car on pourrait imaginer de donner comme résultat, soit :

abs (val (X))

soit absfl (fxfl (val (X))).

Cependant, dans le premier cas, on a appliqué :

C (L2, val)

et dans le deuxième :

C (L2, val) ⁂ C (L1, fxfl) = C (L4, fxfl • val) (cf. 4.5.3.2.)

Or la procédure ABS a deux membres : un imposant L2, l'autre L3 ; de ce fait la deuxième solution est rejetée (i (L3, L4)) et il ne subsiste que la première solution, donnant un résultat ENTIER.

(2) - Le problème est plus délicat pour les opérations binaires : PLUS, etc.... En effet, avec :

apply (PLUS, X, Y)

il faut autoriser les cas où X et Y sont respectivement (ENTIER et ENTIER) ou (REEL et REEL), mais aussi (ENTIER et REEL) ... (REFENTIER et REEL) .... Dans ce dernier cas, on doit pouvoir appliquer sur REFENTIER les conversions C (L2, val) ⁂ C (L1, fxfl), et de même si l'on a (REEL et REFENTIER), mais les deux dernières possibilités amèneraient une ambiguité pour le cas (REFENTIER, REFENTIER), car on aurait alors :

- C (L2, val) et application de plus (val (X), val (Y))
- C (L2, val) ⁂ C (L1, fxfl) et application de
  plusfl (fxfl (val (X)), fxfl (val (Y)))

On est donc obligé d'introduire une technique de "balancing" comme en Algol 68, avec les deux possibilités de niveaux, cart (L4, L3) et cart (L3, L4) pour éviter d'appliquer la conversion C (L1, fxfl) sur les deux arguments du PLUS. Evidemment, cela conduit à une double solution identique pour le couple (REEL, REEL) car on peut passer soit par L5, soit par L6, mais ceci n'a pas d'importance puisque la génération est la même dans les deux cas. D'ailleurs, l'introduction du niveau L = union (L5, L6) résout ce problème.

6.6.5. Extensions syntaxiques

Les extensions syntaxiques doivent porter essentiellement sur l'écriture des expressions arithmétiques. On présente ici un ensemble de règles qui introduisent les opérateurs usuels : +, -, *, •|• (pour /), ** ainsi que || pour valeur absolue, ! pour factorielle, l'opérateur de comparaison > et l'opérateur d'affectation :=. Ces opérateurs ont leur priorité habituelle, et on peut modifier l'ordre d'évaluation en groupant une sous-expression entre parenthèses. Voici deux exemples traduits automatiquement en langage de base par l'analyseur de Ph. Chatelin [8]. L'expansion fait apparaître l'opérateur APPLY puisque le programme généré est ensuite traité par le mécanisme des classes pour appliquer les conversions éventuelles.

```
x:=a**b+c!-d*|e+f-g| ;
```

```
APPLY ( ASSIGN , X , APPLY ( MOINS , APPLY ( PLUS , APPLY ( EXP , A ,
B ), APPLY ( FACT , C )), APPLY ( MULT , D , APPLY ( ABS , APPLY (
MOINS , APPLY ( PLUS , E , F ), G )))));
```

```
(((a+b-c)))>|x| ;
```

```
APPLY ( SUP , APPLY ( MOINS , APPLY ( PLUS , A , B ), C ), APPLY (
ABS , X ));
```

Les règles de transformation reprennent un petit sous-ensemble de la syntaxe de base et décrivent l'arbre des expressions d'une manière classique (pour le formalisme, cf Annexe 4).

```
U ->  SSEXP ';' ;
EXP -> EXP LEXP |ID ;
LEXP -> '(' SSEXP ')' ;
SSEXP -> EXP | EXP ',' SSEXP ;
SSEXP -> AA.1 AD.1 = (AA.1 -> AB |(AB -> AE |
         (AE -> AT |(AT -> AF |(AF -> AG |
       (AG -> AH |(AH -> '(' AA.2 ')' |
         (AD.1 -> ')' |'$A$' AA.2 ')' | '$A$' AA.2 ';') |
        (AH -> '|' AA.2 '|' |(AD.1->')' |'$H$' AH ')'|
                        '$H$' AH ';')|¬))
        |(AD.1 -> ')'|'$G$' AG ')' | '$G$' AG ';' ))
       |(AD.1 -> ')'|'$F$' AF ')' | '$F$' AF ';'))
        |(AD.1 -> ')' | '$T$' AT ')' | '$T$' AT ';' ))
        |(AD.1 -> ')' | '$E$' AE ')' |'$E$' AE ';' ))
        |(AD.1 -> ')' | '$B$' AB ')' | '$B$' AB ';' ))
        |(AD.1 -> ')' | '$A$' AA.1 ')' | '$A$' AA.1 ';' )) ;
AD -> ';' | ')' ;
AA -> ID ':=' AA | AB ;
AB -> AB '>' AE | AE ;
AE -> AE OA AT | AT ; OA -> '+' | '-' ;
AT -> AT OM AF | AF ; OM -> '*' | '.|.' ;
AF -> AF '**' AG | AG ;
AG -> AG '!' |AH ;
AH -> EXP | '(' AA ')' | '|' AA '|' ;
SSEXP -> '$A$' AA.1 =
      (AA.1 -> ID ':=' AA.2 |'APPLY(ASSIGN,' ID ',$A$' AA.2 ')' |:
        AA.1 -> AB |'$B$' AB )|
     '$B$' AB.1 =
      (AB.1 -> AB.2 '>' AE |'APPLY(SUP,$B$' AB.2 ',$E$' AE ')' |:
      AB.1 -> AE |'$E$' AE ) |
      '$E$' AE.1 =
     (AE.1 -> AE.2 OA AT |(OA -> '+' |'APPLY(PLUS,$E$' AE.2 ','
             '$T$' AT ')'
         |:OA -> '-' |'APPLY(MOINS,$E$' AE.2 ',' '$T$' AT ')')
        |: AE.1 -> AT |'$T$' AT )
      |'$T$' AT.1 =
    (AT.1 -> AT.2 OM AF |(OM -> '*'|'APPLY(MULT,$T$' AT.2 ',$F$' AF ')'
         |: OM ->  '.|.' |'APPLY(DIV,$T$' AT.2 ',$F$' AF ')')
        |: AT.1 -> AF | '$F$' AF )
     | '$F$' AF.1 =
       (AF.1 -> AF.2 '**' AG | 'APPLY(EXP,$F$' AF.2 ',$G$' AG ')'
    |: AF.1 -> AG |   '$G$' AG )
     | '$G$' AG.1 =
      (AG.1 -> AG.2 '!' | 'APPLY(FACT,$G$' AG.2 ')' |:
       AG.1 ->  AH |'$H$' AH )
     |'$H$' AH =
       (AH -> EXP |EXP |: AH -> '('AA ')' | '$A$' AA
            |:AH ->  '|' AA '|' |'APPLY(ABS,$A$' AA ')' ) ;
ID -> *L1 ;
```

6.6.6. Utilisation des classes

On voit bien l'utilisation des classes pour convertir implicitement un REFENTIER en ENTIER ou un ENTIER en REEL. Il est plus difficile de se faire une idée de l'utilisation des classes PAIR ou IMPAIR. Voici, à titre d'exemple, un cas que l'on peut traiter par les classes et qui ne peut pas l'être dans les autres langages évolués par les modes ou les attributs. Il s'agit de la fonction DZETA de Riemann dont la formule est, dans le cas général :

$$(1) \text{ - dzeta } (r) = \sum_{n=1}^{\infty} \frac{1}{n^r}, \quad r > 1$$

On ne s'intéresse qu'aux valeurs entières de r. Dans le cas particulier où r est pair, la formule est équivalente à :

$$(2) \text{ - dzeta } (r) = \frac{1}{2} \mid B\ (r) \mid (2\Pi)^r / r\ ! \qquad r > 1$$

où B(r) est le nombre de Bernouilli d'indice r. Il suffit donc d'écrire deux modules que l'on transforme en opérateur, l'un idzeta utilisant la formule (1), l'autre, pdzeta utilisant la formule (2). La déclaration de la procédure générique DZETA est :

```
def (DZETA, proc (L, PAIR, REEL, pdzeta (fconst))),
def (DZETA, proc (L, IMPAIR, REEL, idzeta (fconst)))
```

Après quoi, on peut avoir les déclarations :

```
def (TROIS, as (IMPAIR, 3))
def (DEUX, as (PAIR, 2))
```

et une expression du genre :

```
TROIS * (apply (DZETA, TROIS) + apply (DZETA, DEUX))
```

qui sera traduite, après traitement des classes en :

```
multfl (fxfl (3), plusfl (idzeta (3), pdzeta (2)))
```

CONCLUSION

A la fin de ce travail, il me semble important d'oublier un peu le côté formel du langage puisqu'il sert uniquement de support à plusieurs idées que je voudrais rappeler.

La première est la notion de module qui contient à la fois un ensemble d'information et le traitement de cette information (accès, opérations, ...). Le besoin de l'introduction de tels éléments se fait déjà sentir depuis quelque temps et apparaît dans certains nouveaux langages (Bliss [51]).

Associé aux modules, le procédé de l'actualisation semble fondamental. En effet, c'est un pas de plus vers une certaine liberté dans l'utilisation de la mémoire : Fortran IV fait des allocations statiques ; Algol 60, des allocations par blocs ; PL/1 et Algol 68, l'allocation de certaines variables sur un "tas" (ALLOCATE ou générateurs) ; ici ce sont toutes les variables d'un module qui sont allouées par l'utilisateur. Le mode d'allocation doit alors être général (peut-être en liste) avec optimisation en pile lorsque certaines conditions du flot de contrôle sont réunies (imbrication dynamique des opérateurs act et desact). Le fait de compliquer l'allocation et donc de ralentir les performances n'est pas purement gratuit mais s'impose dans le cas où l'on veut utiliser les facilités de passage de contrôle d'un module à un autre que ce soit avec un seul index actif (principe des coroutines) ou plusieurs (parallélisme).

Un autre point important est l'extension de l'interpréteur, et plus généralement l'extension "sémantique". Il est impossible, lors de la définition d'un langage, même de haut niveau, de donner tous les opérateurs primitifs nécessaires à toutes les applications possibles de l'ordinateur, comme par exemple, le traitement graphique, la simulation, etc.... La tentative de définir ces opérateurs primitifs en termes du langage existant est source de complications et de difficultés. Voilà pourquoi, en possédant dans le langage un mécanisme d'intégration de fonc-

tions sémantiques de bas niveau (opérateur op) on se donne une liberté illimitée d'extension avec une efficacité qui semble satisfaisante. Evidemment, cela ne peut être fait que par des spécialistes qui connaissent les conventions d'écriture pour intégrer un nouvel opérateur à l'interpréteur, mais le travail de programmation est à coup sûr plus léger que celui d'implémenter tout un langage pour traiter une nouvelle application. De plus, la composition (opcomp) et la compilation interne (opmod) assurent une sorte de génération télescopique ("bootstrapping") du système.

Le point qui me semble le plus nouveau est celui des primitives du contrôle d'exécution. Elles pourraient être utilisées d'une façon beaucoup plus souple (mélange syntaxique des expressions et du contrôle ; suppression du next, ce qui sous-entend une exécution séquentielle des unités, etc...) mais j'ai voulu les séparer par souci de clarté et pour bien analyser leur action ; je reviendrai aussi sur leur intérêt théorique.

Il faut souligner aussi deux méthodes qui rentrent dans le cadre des "compilateurs de compilateurs", je veux parler de l'extension syntaxique par macros qui permet une modification aisée de la syntaxe du langage avec modification interne de l'analyseur [8] et qui s'avère un outil puissant de réduction de langages, et ensuite du mécanisme des classes qui génère une vérification de type à partir de définitions données formellement.

Cela m'amène à parler des perspectives qui semblent encore ouvertes. En effet, le problème du contrôle des données à la compilation a été abordé dans les langages sous plusieurs formes ; je citerai quatre orientations possibles :

- modes de base et constructeurs (Algol 68 [45] - EL1 [47])
- domaines de valeurs et applications sur les domaines (Pascal [48])
- définitions de données et utilisation (Classes de Simula 67 [12])
- classes et opérations d'ensemble (chap. 4) [25].

Ces différentes méthodes ne recouvrent pas les mêmes possibilités ; un travail intéressant serait d'étudier leurs champs d'application et d'essayer d'en déduire une synthèse ou une généralisation.

Une autre perspective est la formalisation du flot de contrôle [1] à partir des primitives. Un plan de travail est déjà prévu pour cette étude [5] et il consiste à représenter les modules sous forme de graphes où les noeuds sont les expressions, et les arcs, les relations de branchement (move et next) relativement au même index ; les autres primitives affectent aux noeuds des propriétés particulières. Il semble que ce formalisme puisse apporter des résultats sur la validité de la structure des programmes et par le passage aux schémas de programmes équivalents [15] dériver sur des études plus théoriques telles que transformations de programmes [18] ou preuves de programmes [17]. Ce sont, bien sûr, des problèmes où il est difficile de dire si le travail sera suivi de résultats ou même si cette direction de recherche est valable.

Enfin, toujours dans un esprit de prospective, il serait bon, à la lumière de la séparation entre le contrôle d'exécution et l'évaluation des expressions de repenser le problème de la définition sémantique des langages. Je crois que pour avancer sur ce sujet, il faut raisonner davantage en terme d'automate qu'en terme de syntaxe, si abstraite soit-elle. Les méthodes issues de la syntaxe ont fait progresser énormément le problème de la sémantique [28] [46] ; je m'en suis moi-même servi au chapitre 5. Mais il me semble qu'elles ne sont pas entièrement satisfaisantes pour résoudre deux problèmes majeurs : les effets de bord et le sens des identificateurs, et surtout les changements d'environnement provenant du contrôle d'exécution. On a essayé de trouver des solutions en réduisant ces deux points (programmation sans goto ; suppression des variables [2]) mais ce ne sont que des essais ponctuels. Une autre méthode consisterait au contraire à intégrer le contrôle à la sémantique, d'où l'aspect automate ou algébrique dont je parlais plus haut. La voie mérite à mon avis d'être essayée [38].

ANNEXE 1

On donne les fonctions primitives de l'interpréteur qui semblent nécessaires à une implémentation. L'interpréteur étant extensible, il n'est pas indispensable de s'en tenir uniquement à cette liste ; l'utilisateur peut ne vouloir en garder qu'un sous-ensemble, ou au contraire en demander d'autres suivant le domaine dans lequel il travaille.

1 - SELECTION DANS LES OBJETS STRUCTURES

Opérateur	Objet argument	Objet résultat	Champ sélecté
orig	reference	adresse	premier
tref	reference	déplacement	second
rf	adresse	reference	premier
tadr	adresse	déplacement	second
f	déplacement	constante type fx	premier
u	déplacement	constante type car	second

2 - TRANSFORMATIONS DE CONSTANTES

Les constantes sont caractérisées par un certain type de codage interne. Ce codage est dépendant de la machine. D'après 2.2.1. § 4, nous avons choisi les codages :

car bit hexa fx(virgule fixe) fl(virgule flottante)

L'interpréteur comprend les fonctions de passage d'une représentation interne à une autre — dans la mesure où l'on peut trouver un élément correspondant. Le tableau suivant donne ces fonctions qui prennent un argument et retournent un résultat.

Résultat / Argument	car	bit	hexa	fx	fl
car	-	carb	carh	carfx(*)	carfl(*)
bit	bitc(*)	-	bith	bitfx	bitfl
hexa	hexc(*)	hexb	-	hexfx	hexfl
fx	fxc(*)	fxb	fxh	-	fxfl
fl	flc(*)	flb	flh	flfx(*)	-

Les opérateurs marqués d'une (*) ne sont pas toujours définis.

Exemple :

carfx (car (123)) est défini ;

car (123) a comme représentation (IBM/360) : F1F2F3 hexadécimal et carfx (car (123)) est un entier en virgule fixe qui vaut 123 en base 10.

Par contre carfx (car (ABC)) n'est pas défini et provoque une erreur de conversion. Si l'on veut obtenir la valeur numérique du codage des caractères, il faut une transformation intermédiaire : transformer les caractères en hexadécimal, et ensuite en virgule fixe, pour le cadrage.

d'où : hexfx (carh (car (ABC)))

On a alors comme résultat la valeur C1C2C3 en base 16, c'est-à-dire 12 698 307 en base 10.

3 - FONCTIONS GENERALES D'UTILISATION DES OBJETS

Ce sont les opérateurs que l'on emploie habituellement dans les expressions. On les donne sous forme de tableau ; le sens est en général connu.

OPERATIONS SUR LES CONSTANTES

Arithmétiques

Opérateur	nbre d'opérandes	type opérandes	type résultat
plus	2	fx	fx
moins	2	fx	fx
mult	2	fx	fx
div	2	fx	fl (division arithmétique)
diventier	2	fx	fx (division entière)
exp	2	fx	fx
reste	2	fx	fx (reste de la division entière)
plusfl	2	fl	fl
moinsfl	2	fl	fl
multfl	2	fl	fl
divfl	2	fl	fl
expfl	2	fl, fx	fl
abs	1	fx	fx (valeur absolue)
pl	1	fx	fx (plus unaire)
ms	1	fx	fx (moins unaire)
absfl	1	fl	fl (valeur absolue, plus et moins unaires pour flottant).
plfl	1	fl	fl
msfl	1	fl	fl

Caractères

conc	2	car	car (concaténation)
sch	3	car, fx, fx	car (sous-chaîne)

Chaînes de bits

ou	2	bit	bit (union de bits)
et	2	bit	bit (intersection)
ni	2	bit	bit (ou exclusif)
non	1	bit	bit (complément)

Prédicats

Les prédicats sont assimilés aux expressions ; leur résultat est booléen : true ou false et dans les expressions, il se traduit par les chaînes : bit (1) ou bit (0). Donc, une expression sur un bit peut servir de prédicat dans les conditionnelles.

Les opérateurs de comparaison que l'on donne sont tous binaires.

Opérateur	type des opérandes	
inf	fx	<
eq	fx	=
sup	fx	>
infeg	fx	⩽
supeg	fx	⩾
inffl	fl	<
eqfl	fl	=
supfl	fl	>
infegfl	fl	⩽
supegfl	fl	⩾

Remarques :

(1) Si l'on a des constantes de types différents, il faut utiliser les fonctions de transformation de 2.

Exemple : def (A, fx (4))

multfl (fl (3), fxfl (A))

(2) Pour obtenir des prédicats sur d'autres types de constantes, on doit faire des compositions avec les fonctions de transformation.

Exemple : comparaison de chaînes de caractères :

```
def (EQCAR, opcomp (eq (hexfx (carh (fconst)), hexfx (carh
                                                (fconst)))))
```

qui permet d'utiliser : eqcar (car (A), car (B))

Opérations sur les références

val 1 argument : référence résultat : objet

store 2 arguments : référence, objet pas de résultat

eqref prédicat d'égalité de deux références.

Opérations sur les adresses

compadr 2 arguments : adresse, déplacement résultat : adresse.

sousadr 2 arguments : adresse, déplacement résultat : adresse.

compadr ajoute le déplacement à l'adresse.

sousadr retranche le déplacement, dans les limites de la zone où l'on se trouve.

eqadr prédicat d'égalité de deux adresses.

infadr, supadr } prédicats de comparaison des adresses.

Opérations sur les déplacements

addep, sdep } 2 arguments : déplacement résultat : déplacement

addep ajoute deux déplacements avec les conventions de cadrage.

sdep retranche le deuxième déplacement au premier, qui doit lui être supérieur.

eqdep, infdep, supdep } prédicats de comparaison des déplacements.

Opération sur les modules, locus, opérateurs

Actualisation des modules : act (cf. 2.2.3.2.)

Allocation des locus : alloc (cf. 2.2.3.3.)

Désactualisation : desact (cf. 2.2.3.6.)

Liaison des paramètres : bind (cf. 2.2.3.4.)

Pour les opérateurs, on peut considérer que op définit une dénotation — au même titre que ref, adr, mod ... — et que opmod et opcomp sont des fonctions de l'interpréteur.

Fonctions particulières

Relativement aux index, on a la fonction ret (cf. 3.3.1.1. §5) qui donne une liste d'index. Les autres fonctions (activate, cond, next ...) sont des fonctions du moniteur et non de l'interpréteur. Cette distinction est cependant limitée par l'implémentation ; dans la mesure où l'on ne peut pas compiler les fonctions de contrôle sur l'unité de contrôle de la machine, on a besoin d'un deuxième interpréteur, que l'on appelera interpréteur de contrôle et qui aura pour tâche de simuler le moniteur.

L'évaluation des expressions nécessite une pile d'évaluation. On a accès à cette pile par les fonctions :

stack (liste d'arguments) - qui met les arguments sur la pile, en commençant par le dernier

unstack (liste de references) - enlève les éléments de la pile et les range dans les références de telle sorte que l'on a :

stack (fx (1), fx (2), fx (3))
unstack (A), unstack (B, C)

alors : A contient fx (1)
B contient fx (2)
C contient fx (3)

decap (<expression>) - l'expression donne un nombre entier n et l'opérateur enlève n éléments du sommet de pile.

4 - FONCTIONS PROPRES AUX ENTREES-SORTIES

Ces fonctions dépendent entièrement de la configuration de la machine sur laquelle on travaille. On peut proposer les fonctions suivantes (en exemple) :

lirecarte (référence)	lit une carte en entrée et range le contenu dans la référence en paramètre
imprimer (constante car)	imprime la chaîne de caractères sur une ligne d'imprimante.
sautligne (n)	saute n lignes à l'imprimante.
lireligne (référence)	lit une ligne au terminal et range le contenu dans la référence.
écrireligne (constante car)	envoie la chaîne de caractères sur le terminal.

Si l'on veut utiliser des disques, bandes, ou autres unités, il faut des fonctions plus complètes pour l'ouverture et la fermeture des fichiers, l'accès non séquentiel, etc....

Nous ne rentrons pas dans ces détails, mais la méthode utilisée jusqu'ici donne la façon dont on peut procéder.

A N N E X E 2

SYNTAXE CONCRETE DU LANGAGE DE BASE

METASYNTAXE

(0-1) - <suite de TAG1s> → <TAG1> <suite de TAG1s> | <TAG1>

(0-2) - <TAG1> → <definition de module> | <unité> | <digit> | <car>

(0-3) - <liste de TAG2s> → (<suite séparée de TAG2s>)

(0-4) - <suite séparée de TAG2s> → <TAG2> , <suite séparée de TAG2s>|<TAG2>| ε

(0-5) - <TAG2> → <index> | <defexprsimple> | <expression> |
<definition-c> | <identificateur de mod> | identificateur local |
<contrôle> | <alternative> | <interruption>

SYNTAXE 1

(1) - <programme> → <suite de définitions de module> | <execution>

(2) - <définition de module> → def (<identificateur de mod>, <module littéral>)

(3) - <exécution> → execute (<index>, <module>, <suite séparée d'expressions>)

(4) - <module> → <module littéral> | <identificateur de mod>

(5) - <unité> → (<etiquette>, <defexpr>, <contrôle>)

(6) - <etiquette> → identificateur local | ε

(7) - <identificateur de mod> → identificateur local

(8) - <defexpr> → <liste de defexprs simples> | <defexpr simple> | ε

(9) - <defexpr simple> → <definition> | <expression>

(10) - <expression> → <expression> <liste d'expressions> | <objet de base> |
<identificateur>

(11) - <définition> → def (identificateur local, <expression>) |
use <liste d'identificateurs de mod>

(12) - <identificateur> → identificateur local | <identificateur qualifié>

(13) - <identificateur qualifié> → identificateur local • <identificateur>

(14) - <objet de base> → <objet litteral> | <objet formel>

(15) - <objet litteral> → ref (<expression>, <expression>) |
adr (<expression>, <expression>) |
dep (<expression>, <expression>) |
<constante littérale> |
<module littéral> |
<operateur littéral> | locus | signal

(16) - <objet formel> → fref | fadr | fdep | fconst | fmod | fop | floc | fsig

(17) - <module litteral> → mod (<graphe des classes>, <liste d'index>,
<suite d'unités>)

(18) - <opérateur litteral> → op (corps de base) |
opcomp (<expression>) |
opmod (<expression>, <index>, <result>)

(19) - <result> → stack <liste d'identificateurs locaux> | ε

(20) - <constante littérale> → car (<suite de cars>) |
bit (<suite de digits>) |
hexa (<suite de cars>) |
fx (<suite de cars>) |
fl (<suite de digits> • <suite de digits>,
<suite de digits>)

(21) - <index> → <identificateur> | * |
ret (<index>, <suite séparée d'expressions>)

(22) - <car> → <digit> | A | B | C | D | E | F | G | H | I | J | K | L ... | Z

(23) - <digit> → 0 | 1 | 2 | 3 | ... | 9

(24) - <contrôle> → <contrôle composé> | <contrôle conditionnel> | <primitive> | ε

(25) - <contrôle composé> → ctl <liste de contrôles>

(26) - <contrôle conditionnel> → cond <liste d'alternatives>

(27) - <alternative> → (<expression>, <contrôle>)

(28) - <primitive> → activate (<index>, [<paramètres d'activation>]*) |
stop <liste d'index> | resume <liste d'index> |
kill <liste d'index> | next (<index>) |
move (<index>, <etiquette>) |
wait (<identificateur>, <liste d'index>, <contrôle>) |
free (<identificateur>, <liste d'index>)

(29) - <paramètres d'activation> → entry : <etiquette> | etat : <etat> |
interrupt : <liste d'interruptions> | pty : <expression> |
ret : <liste d'index>

(30) - <etat> → pret | attente

(31) - <interruption> → st | none | <type> (<module>, <index>, <suite séparée d'expressions>)

(32) - <type> → zerodivide | overflow ...

SYNTAXE 2

(33) - <graphe des classes> → declass <liste de definition-c> | <identificateur de mod> | ε

(34) - <definition-c> → def (<id de relation>, <relation>)

(35) - <relation> → <expression-c-l> | <proc> | <conv>

(36) - <expression-c-l> → class | level | in identificateur **local** |
union <liste d'identificateurs locaux> |
inter <liste d'identificateurs locaux> |
compl (identificateur local, identificateur local)
cart <liste d'identificateurs locaux>

(37) - <proc> → proc (identificateur local, <class>, <class>, <expression>)

(38) - <conv> → conv (identificateur local, <class>, <class>, <expression>)

(39) - <class> → identificateur local | cart <liste d'identificateurs locaux>

(40) - <id de relation> → identificateur local | ε

(10)* - <expression> → <objet classifié> | apply (identificateur local,
<suite séparée d'expressions>)

(41) - <objet classifié> → as (identificateur local, <expression>)

La règle (10)* est le complément de la règle (10) dans le cas de l'utilisation des classes.

ANNEXE 3

ARTICULATION SOFTWARE-HARDWARE

On a remarqué que, dans un langage sans type, certains choix de définition dépendent de la machine. Plutôt que d'ignorer cette dépendance, il faut au contraire la mettre en évidence pour bien la délimiter. La "portabilité" du compilateur se fera en réécrivant les parties du système qui font "l'articulation software-hardware". On peut néanmoins obtenir des softwares "indépendants de la machine" à partir du langage de base, soit en passant par le mécanisme des classes, soit simplement en écrivant un prélude de définitions qui cache l'aspect de la machine à l'utilisateur.

Exemple : Une allocation de variables se fait avec la fonction de base mem. On a vu comment réaliser une certaine organisation de la mémoire par l'opérateur taille. Si l'on veut aller plus loin et supprimer — pour l'utilisateur — le souci de préciser les paramètres du déplacement (dépendant de la machine), on écrira les extensions de grammaire (formalisme [8]) :

```
<définition> → def (<identificateur>, <motclé>) =
      (<motclé> → reference | def (<identificateur>, taille (1, 'W'))
    |:<motclé> → pointeur | def (<identificateur>, taille (f (DEPREF),
                                                           u (DEPREF)))
    |:<motclé> → caractère | def (<identificateur>, taille (1, 'BYTE'))| )
```

et on peut écrire :

(def (A, reference), def (P, pointeur), def (C, caractère))

Pour une machine ayant des mesures de déplacement différentes, il suffit de changer la partie remplacement des règles d'extension de la définition et le programme pourra être exécuté sur cette nouvelle machine sans modifications.

Voici les définitions à modifier ou compléter suivant la machine :

TRAITEMENT DES DONNEES

Syntaxe

(1) - <déplacement littéral> → suivant qu'il s'agit d'une machine à mots, à caractères ....

(2) - <constante littérale> → suivant les types de codages présents dans la machine.

Opérateurs

- Demande de mémoire centrale : mem
- Opérations élémentaires suivant les types de codage, en liaison avec la règle (2).

Exemple :

codage	type d'opération
bit	opérations booléennes
octal hexadécimal	opérations sur les zones
virgule fixe virgule flottante décimal	opérations arithmétiques
caractère	opérations sur les caractères

Remarque : Il est bien entendu que les transformations de représentation interne permettent d'appliquer n'importe quelle opération sur n'importe quel type de constante, ou à peu près, et que, la chaîne de bits englobant tous les types de codages, on peut faire des opérations booléennes sur un nombre codé en décimal, mais que ce ne sera pas l'emploi le plus fréquent.

- Cadrage des données : on le réalise en définissant les déplacements pour les types d'objets pouvant être mis en mémoire :

  DEPDEP déplacement d'un déplacement.
  DEPADR déplacement d'une adresse.
  DEPREF déplacement d'une référence.
  DEPMOD déplacement d'un module.
  . (d'un descripteur de module formel).
  .
  .

- Opérations d'entrée-sortie.

  Elles dépendent non seulement de la machine, mais de la configuration des périphériques. Il est bien connu que la portabilité d'un software passe par la réécriture des procédures d'entrée-sortie.

CONTROLE D'EXECUTION

Toutes les primitives du contrôle d'exécution doivent être compilées ou interprétées, mais elles font partie du langage. Par contre, les interruptions dépendent de la machine, et s'insèrent à la fois au niveau syntaxique et au niveau du traitement à apporter.

Syntaxe

(3) - <type> → mots-clé des interruptions synchrones.

Opérateurs

A chaque type d'interruption correspond un opérateur du traitement de cette interruption (cf.3.5.1).

ANNEXE 4

FORMALISME DES MACROS-SYNTAXIQUES

Les idées de T.E. Cheatham sur les S-macros [9] sont à l'origine du travail de S.A. Schuman [36] qui définit une méthode pour modifier l'arbre de l'analyse syntaxique au moyen de règles conditionnelles de transformation. Ph. Chatelin et B. Willis ont étudié et réalisé l'implémentation d'un tel système [8] en utilisant le principe d'analyse de J. Earley [14]. La grammaire d'entrée doit s'écrire sous la forme suivante :

(1) - Les règles de grammaire sont écrites à l'aide des métasymboles '→' pour une dérivation et '|' pour séparer les alternatives. La fin de règle est le ';'. Par convention, la partie gauche de la première règle est considérée comme l'axiome. Les symboles terminaux sont écrits entre apostrophes.

Exemple :

S → 'b' A 'b' ;

A → 'd' | 'a' A 'a' | 'c' A 'c' ;

Cette grammaire génère les phrases du type

(a) b w d $\tilde{w}$ b

où $\tilde{w}$ est l'image miroir de w et w est la chaîne vide ou une chaîne quelconque composée de 'a' et de 'c'.

(2) - Le remplacement simple s'écrit à la suite d'une règle, à l'aide du symbole '='. Si l'on écrit :

$$X \rightarrow \alpha = \beta \ ;$$

cela signifie que lorsque l'analyseur aura reconnu une occurrence de la règle $X \rightarrow \alpha$, il remplacera le sous-arbre issu de X par la chaîne β et recommencera l'analyse de cette chaîne à partir de X. La chaîne α et la chaîne β peuvent contenir des non-terminaux, mais les non-terminaux de β doivent avoir été reconnus lors de l'analyse de α et il y a alors substitution de ces non-terminaux et des sous-arbres correspondants au cours du remplacement. En reprenant l'exemple précédent, on introduit le remplacement :

S → 'b' A 'b' ;

A → 'd' | 'c' A 'c' | 'a' A 'a' = 'c' A 'c' ;

L'analyseur reconnaît bien les phrases du type (a) pour cette grammaire , mais après analyse, ces phrases sont remplacées par :

(b) $\quad bc^n dc^n b$

(3) - Le remplacement conditionnel s'écrit sous la forme :

$$X \rightarrow \alpha = (P \mid R1 \mid R2) \ ;$$

qui signifie : si le prédicat P est vrai, alors remplacement R1, sinon remplacement R2. R1 et R2 peuvent eux-mêmes être des remplacements conditionnels. Deux symboles peuvent apparaître en position de remplacement :

'=' qui signifie que l'on ne modifie pas le sous-arbre.

'¬' qui signifie que l'on ne continue pas l'analyse avec cette règle et qu'il faut alors essayer une autre alternative.

Les prédicats sont écrits sous forme de règles et indiquent quelle doit être la forme du sous-arbre issu de la catégorie X pour effectuer la transformation.

Exemple :

X → 'b' | 'c' ;

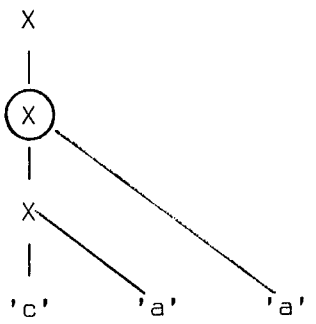
X → X 'a' = (X → 'c' | 'ba' | =)

Le remplacement se lit : si X donne 'c', alors remplacer par la chaîne 'ba' sinon continuer l'analyse.

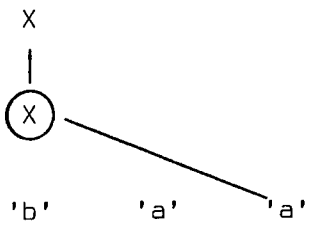
Ainsi la chaîne :

'caa'

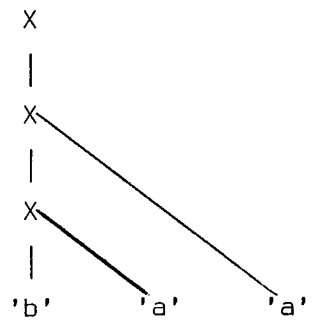
a pour arbre d'analyse :



La macro s'applique sur le X entouré et remplace le sous-arbre par :



L'analyse reprend à partir de ce X, ce qui donne :



La chaîne résultante est 'baa'.

D'autres symboles sont introduits dans le calcul des prédicats :

(P1 |: P2 | R1 | R2)

signifie : (P1 | (P2 | R1 | R2) | ¬)

c'est le "alors si" ou le "sinon si".

On a utilisé aussi :

(P1 |+ P2 | R1 | R2)

qui signifie : (P1 | R1 | (P2 | R1 | R2))

c'est le "ou si".

B I B L I O G R A P H I E

[1] - F. E. ALLEN
Control Flow Analysis - Sigplan Notices - July 70 -

[2] - J. BACKUS
Reduction Languages and Variable - free programming -
IBM Research Laboratory - San Jose - California -

[3] - N. BEKIĆ and K. WALK
Formalization of storage properties - Symposium on semantics of Algorithmic Languages - Ed. New-York (1971) -

[4] - D. BERT
Bases pour la définition des objets et leurs propriétés et fonctions primitives de contrôle dans les langages extensibles -
Séminaire Mars 1972 - Mathématiques Appliquées - USM Grenoble -

[5] - D. BERT
Formalisation des primitives du flot de contrôle - Notes non publiées -
Mathématiques Appliquées - USM Grenoble -

[6] - D. G. BOBROW and B. WEGBREIT
A model and stack implementation of multiple environments -
Report n° 2334 - Cambridge, Massachusetts -

[7] - R. M. BURSTALL and R. J. POPPLESTONE
POP-2 Reference Manual - Machine Intelligence 2 -
Ed. E. Dale & D. Michie - American Elsevier, New-York (1968) -

[8] - Ph. CHATELIN et B. WILLIS
Le mécanisme des macros syntaxiques - Rapport Scientifique, contrat CRI 70-107 - Décembre 1972 -

[9] - T. E. CHEATHAM
The introduction of definitionnal facilities into higher level programming languages - AFIPS 66 - Fall Joint Computer Conference - vol 19 -

[10] - T. E. CHEATHAM, Jr. A. FISHER, Ph. JORRAND
On the basis for ELF, an extensible language facility - Proc. of AFIPS - 1968 - Fall Joint Computer Conference, vol 33 - Part. 2 -

[11] - M. E. CONWAY
Design of a separable transition - diagram compiler - C. ACM-6 (1963) -

[12] - O. J. DAHL, B. MYHRHAUG, K. NYGAARD
Simula 67, Common base language - Norwegian Computing Center - N° S-2-

[13] - E. W. DIJKSTRA
Cooperating sequential processes - Programming Languages - (Ed. Genuys) - Academic Press - New-York 68 -

[14] - J. EARLEY
An efficient context - free parsing algorithm
C.ACM - 13-2 - Feb. 70 -

[15] - A. P. ERSHOV
Theory of program schemata - IFIP congress - Ljubljana - Août 71 -

[16] - R. W. FLOYD
Non deterministic Algorithms - C. ACM - 11-9-Sept 68 -

[17] - R. W. FLOYD
Assigning meanings to programs - Proc. Symposium Applied Mathematics vol 19 - 1967 -

[18] - S. J. GARLAND, D. C. LUCKHAM
Translating Recursion Schemes into Program Schemes -
Sigplan Notices - vol 7 - N° 1 - Janv. 72 -

[19] - J. V. GARWICK
GPL, a truly general purpose language - C.ACM - 11-9 - Sept. 68 -

[20] - G. H. HOLLOWAY
Interpreter - Compiler integration in ECL - Proc. of the International Symposium on Extensible Languages - Sigplan Notices - vol 6 - n° 12 - Déc. 71 -

[21] - S. W. GOLOMB, L. D. BANNET
Backtrack Programming - J. ACM - 12 Oct. 65 -

[22] - W. HENHAPL
A storage model derived from axiom
TR. 25100 (1969) - IBM Laboratory Vienna

[23] - E. T. IRONS
Experience with an extensible language : IMP - C.ACM - 13-1-Janv. 70 -

[24] - Ph. JORRAND
Data types and extensible languages - Sigplan notices - vol 6 - n° 12 - Déc. 71 -

[25] - Ph. JORRAND and D. BERT
On some basic concepts for extensible programming languages - International Computing Symposium - Venise - April 72 -

[26] - Ph. JORRAND and D. BERT
Some bases for control structure and environment in programming languages - (en préparation) -

[27] - B. M. LEAVENWORTH
Syntax macros and extended translation - C.ACM - 9-11-Nov. 66 -

[28] - P. LUCAS and all
Method and notation for the formal definition of programming languages - TR. 25 085, IBM Laboratory Vienna - Déc. 68 -

[29] - J. Mc CARTHY
Lisp 1.5 Programmer's Manual - The MIT Press (1962)

[30] - M. D. Mc ILROY
Macro instruction extensions of compiler languages - C.ACM - vol 3 - N° 4 - April 60 -

[31] - M. D. Mc ILROY
Coroutines : Semantics in search of a syntax - Oxford University and Bell Telephone Laboratories, Incorporated -

[32] - P. NAUR
Revised report on the algorithmic language Algol 60 - C.ACM - 6.1. (Janv. 63) -

[33] - S. S. PATIL
Coordination of asynchronous events - Ph. D. Thesis, MIT - Sept. 67 -

[34] - Ch. J. PRENNER
The control structure facilities of ECL -
Report n° 2138 (Déc. 71) - Cambridge, Massachusetts -

[35] - J. C. REYNOLDS
Gedanken : A simple typeless language based on the principle of completness and the reference concept -
C-ACM - 13-5 - May 70 -

[36] - S. A. SCHUMAN
Spécification des langages de programmation et de leurs traducteurs au moyen de macros syntaxiques -
Congrès de l'AFCET - N° 2 (1970) -

[37] - S. A. SCHUMAN
An extensible interpreteur - Sigplan Notices - vol 6 - n° 12 - (Déc. 71) -

[38] - D. SCOTT
The lattice of flow diagrams - Symposium on semantics of Algorithmic Languages - Lecture notes in Mathematics - vol. 188 (1971) -

[39] - M. SINTZOFF
Notes non publiées

[40] - N. SOLNTSEFF
A phenomenological classification of extensible programming languages - Department of Applied Mathematics - Mc Master University - Hamilton, Ontario - (July 71) -

[41] - T. A. STANDISH
Some features of PPL, a polymorphic programming language - Proc. of the extensible languages symposium, Sigplan Notices, vol 4 - N° 8 (Aug. 69) -

[42] - T. A. STANDISH
PPL, an extensible language that failed - Sigplan Notices, vol 6 - n° 12 - Déc. 71 -

[43] - D. SUTY
Auto-compilation et conceptualisation des langages de programmation - Thèse USM Grenoble (1971) -

[44] - H.J.M. VAN GILS
Syntactic definition mechanisms - Sigplan Notices - vol 6 - n° 12 - Déc. 71 -

[45] - A. VAN WIJNGAARDEN, B. J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER
Report on the algorithmic language Algol 68 - MR 101 -

[46] - P. WEGNER
The Vienna definition language - Computing Surveys - Vol 4 - N° 1 - Mars 1972 -

[47] - B. WEGBREIT
Studies in extensible programming languages -
ESD-TR.70-297 - Harward University, Cambridge, Massachusetts -
May 70 -

[48] - N. WIRTH
The programming language PASCAL and its design criteria -
NATO Science Committee Conference - Techniques in Software
engineering - Sept 69 - Vol 2 -

[49] - N. WIRTH and H. WEBER
Euler : A generalization of Algol, and its formal definition -
Part I : C.ACM - 9-1 Janv. 66 - Part II - C.ACM - 9-2 Feb. 66 -

[50] - N. WIRTH
On multiprogramming, machine coding, and computer organization -
C.ACM - 12-9 Sept. 69 -

[51] - W.A. WULF, D.B. RUSSELL, A.N. HABERMANN
BLISS : a language for systems programming - C.ACM - 14-12 Déc. 71 -

Vu,

Grenoble, le

Le Président de la Thèse

Vu, et permis d'imprimer

Grenoble, le

Le Président de l'Université Scientifique et Médicale.