Contribution au développement des langages extensibles

Philippe Jorrand

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE

DOCTEUR D'ETAT

Philippe JORRAND

CONTRIBUTION AU DEVELOPPEMENT DES LANGAGES EXTENSIBLES

Thèse soutenue le 28 janvier 1975 devant la Commission d'Examen

Monsieur	N. GASTINEL	Président
Monsieur	M. GRIFFITHS	Rapporteur
Messieurs	L. BOLLIET	
	J.C. BOUSSARD	Examinateurs
	A. van WIJNGAARDEN	

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLY-
TECHNIQUE DE GRENOBLE

M.MICHEL SOUTIF
M.GABRIEL CAU

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

MEMBRES DU CORPS ENSEIGNANT DE L'USMG

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Pédiatrie
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Fraçoise	Mathématiques Pures
MM.	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
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARRAZGilbert	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
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphilligraphie
	FAU René	Clinique neuro-psychiatrique

MM.	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	KAHANE André	Physique générale
	KLEIN Joseph	Mathématiques Pures
	KOSZUL Jean-Louis	Mathématiques Pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques Appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre	Mathématiques Appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques Pures
	MALGRANGE Bernard	Mathématiques Pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Séméiologie médicale
	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
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REULOS René	Physique industrielle
	RINALDI Renaud	Physique
	ROGET Jean	Clinique de pédiatrie et de puériculture
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLANT François	ZOologie
	VALENTIN jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	ASCARELLI Gianni	Physique
	CHEEKE John	Thermodynamique
	GILLESPIE John	I.S.N.
	ROCKAFELLAR Ralph	Mathématiques Appliquées
	WOHLFARTH Erich	Physique du solide

MM.	COHEN-ADDAD Jean Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DELOBEL Claude	M.I.A.G.
	DUSSAUD René	Mathématique (OUS)
Mme	ETEPRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FONTAINE Jean Marc	Mathématiques Pures
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques Appliquées
	GROS Yves	Physique (stag.)
	GROULADE Joseph	Biochimie médicale
	GUITTON Jacques	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KRAKOWIAK Sacha	Mathématique Appliquées
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LEROY Philippe	Mathématiques
	LOISEAUX Jean Marie	Physique nucléaire
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (I.U.T. "A")
Mme	MINIER Colette	Physique
MM.	MICOUD Max	Maladie infectieuses
	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAYNAUD Hervé	MI.A.G.
	RENAUD Maurice	Chimie
Mme	RENAUDET Jacqueline	Bactériologie
M.	RICHARD Lucien	Botanique
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean Claude	Chimie Générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	CRABBEE Pierre	C.E.R.M.O.
	CABOT	Mathématiques Appliquées
	CURRIE Jan	Mathématiques Appliquées

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

MM.	BARGE Michel	Neugo-chirurgie
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculai:
	CORDONNIER Daniel	Néphrologie
	DENIS Bernard	Cardiologie
	KOLODIE Lucien	Hématologie
	RAMBAUD Pierre	Pédiatrie
	ROCHAT Jacques	Hygiène et hydrologie

"MEMBRES DU CORPS ENSEIGNANT DE L'I.N.P.G."

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	ELectrochimie
	BONNETAIN Lucien	Chimie Minérale
	BONNIER Etienne	Electrochimie, Electrométallurgie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	ELectronique
	COUMES André	Radioélectricité
	FELICI Noël	Electrostatique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	SANTON Lucien	Mécanique
	SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M.	BOUDOURIS Georges	Radioélectricité

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BLOCH Daniel	Physique du solide et cristallographie
	COHEN Joseph	Electrotechnique
	DURAND Françis	Metallurgie
	MOREAU René	Mécanique
	POLOUJADOFF Michel	ELectrotechnique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	BOUVARD Maurice	Génie mécanique
	CHARTIER Germain	Electronique
	FOULARD Claude	Automatique
	GUYOT Pierre	Chimie minérale
	JOUBERT Jean Claude	Physique du solide
	LACOUME Jean Louis	Géophisique
	LANCIA Roland	Physique atomique
	LESPINARD Georges	Mécanique
	MORET Roger	Electrotechnique nucléaire
	ROBERT François	Analyse numérique
	SABONNADIERE Jean Claude	Informatique fondamentale et appliquée
Mme	SAUCIER Gabrièle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan Doré	Automatique

CHARGE DE FONCTIONS DE MAITRES DE CONFERENCES

M.	ANCEAU François	Mathématiques appliquées

Fait à Saint Martin d'Hères JANVIER 1974

Je tiens à remercier Monsieur le Professeur GASTINEL, qui me fait l'honneur de présider le jury de cette thèse. Son intérêt pour tout ce qui touche aux applications pratiques des ordinateurs, en particulier à travers l'utilisation des langages de programmation, a suscité entre nous des discussions qui furent parfois passionnées, mais qui furent souvent passionnantes.

Je veux aussi exprimer à Monsieur le Professeur van WIJNGAARDEN ma reconnaissance la plus vive, car ce sont ses travaux sur les langages de programmation, en particulier sur Algol 68, sources d'inspiration pour beaucoup de chercheurs aujourd'hui, qui furent à l'origine de bien des idées et des réflexions que l'on trouvera dans cette thèse. Je le remercie de l'attention avec laquelle il a su juger mon travail.

Je remercie également Michael GRIFFITHS qui m'a toujours accueilli avec bienveillance dans le cadre d'une collaboration fructueuse entre son équipe et les membres du Centre Scientifique IBM de Grenoble dont je faisais partie. C'est là que bon des idées présentées dans cette thèse ont pu se développer. qu'il trouve également dans ces lignes l'expression de ma gratitude pour sa lecture patiente et sa critique précise d'un premier manuscrit qui, je le reconnais, a souvent dû être rebutant.

Lorsque j'étais étudiant à l'Institut Polytechnique de Grenoble en 1961, c'est Monsieur BOLLIET qui m'a fait découvrir ce qu'était la programmation. Aujourd'hui, il a accepté de juger mon travail dont le sujet doit beaucoup à sa clairvoyance. En effet, c'est grâce à lui que j'ai eu la chance de pouvoir travailler aux Etats Unis en 1968 et 1969 dans une equipe de recherche qui fut parmi les premières à explorer le problème des langages extensibles. Pour tout cela, je tiens à lui exprimer ma reconnaissance.

C'est d'un travail avec Jean Claude BOUSSARD que, dès 1966, est né mon intérêt pour la conception et la compilation des langages de programmation. Depuis lors, nos nombreuses rencontres tant à Grenoble qu'au sein de divers groupes de travail français ou internationaux m'ont chaque fois permis de constater que la vigueur, mais aussi l'humour, de ses critiques étaient toujours accompagnés de la sagacité de son jugement. C'est pourquoi je suis heureux qu'il ait bien voulu juger mon travail et je tiens à l'en remercier.

le contenu de cette thèse doit aussi beaucoup à tous ceux avec qui j'ai travaillé depuis quelques années. Mes remerciements vont notamment à Thomas E. CHEATHAM qui fut pour moi un directeur aux idées déterminantes pour l'ensemble de mon travail, et qui a sans cesse été un collègue amical pendant mon séjour aux Etats Unis. Mes remerciements vont aussi à Stephen A. SCHUMAN dont la collaboration m'est toujours précieuse, et à qui je dois de ne jamais perdre de vue l'orientation générale et les objectifs, parfois lointains et mal dessinés, de chaque aspect de mon travail.

Je veux également exprimer ma reconnaissance à la Compagnie IBM France pour l'accueil que le Service de Développement Scientifique, dirigé par Monsieur MOREAU, a fait à mon travail. En particulier, je tiens à remercier Monsieur PELTIER et Monsieur AUROUX pour l'atmosphère chaleureuse qu'ils ont su créer et maintenir longtemps au Centre Scientifique IBM de Grenoble, malgré des circonstances parfois difficiles.

Je remercie enfin vivement Madame CARRE-PIERRAT pour sa patience et son travail de dactylographie remarquablement bien fait, ainsi que tout le personnel du service de reproduction pour son amabilité et sa compétence.

A ma femme,

A mes enfants.

TABLE DES MATIERES

-:-:-:-:-:-:-:-:-:-

.../...

.../...

.../...

.../...

"Chacun les entendait parler sa propre langue ...

Tous étaient stupéfaits et se disaient l'un à l'autre : "que peut bien être cela ?" D'autres encore disaient en se moquant : "Ils sont pleins de vin doux !".

(Actes 2, 6-13)

1. INTRODUCTION

-:-:-:-:-:-:-:-

Pour se servir d'un ordinateur, c'est toujours un langage qui constitue le moyen priviligié. Grâce à lui, l'utilisateur spécifie la nature des informations mises en jeu, leur organisation et les relations logiques ou fonctionnelles de ces informations avec les résultats qu'il veut obtenir.

Si l'on considère, sur un plan historique, l'évolution des langages et des divers mécanismes de nature linguistique qui permettent cette communication avec la machine, on constate d'abord que très tôt, dès la définition de FORTRAN par John Backus (c'était en 1954), l'impulsion a été donnée à un courant fondamental dans la recherche en informatique : le but à atteindre est de fournir à tout utilisateur, de façon aussi économique que possible, des moyens d'expression qui lui soient "naturels", qui lui permettent de construire et d'utiliser facilement et avec sécurité une représentation du domaine concret de l'application qu'il traite, sans pour autant abandonner une certaine rigueur que la programmation impose.

Pour cela, il est d'abord apparu que l'utilisateur doit pouvoir s'abstraire de contraintes spécifiques dues à la structure et au fonctionnement particulier de l'ordinateur, et qui sont totalement étrangères à la nature de son problème. C'est ainsi qu'ont été conçus des langages que l'on a qualifiés d'"évolués" ou "de haut niveau", par opposition au langage machine, ou même à celui d'un macro-assembleur qui, eux, sont "de bas niveau". C'est seulement de ces langages là qu'il sera question dans cet ouvrage. FORTRAN était le premier d'entre eux, puis, surtout après Algol 60, a commencé une véritable inflation des langages de programmation et deux écoles se sont progressivement dessinées. Il y a eu très tôt celle qui défendait l'idée selon laquelle seuls des langages spécialisés par domaine d'application et par type de traitement peuvent vraiment répondre au problème posé. Cette tendance s'est donc surtout souciée de fournir des moyens d'expression "naturels" et adaptés à chaque application, avec pour conséquence inévitable, une croissance incontrôlable du nombre des langages et souvent même de leurs dialectes. La situation a pu alors être comparée avec justesse à celle que nous rapporte la Génese dans l'épisode de la Tour de Babel, d'autant plus que nul souci d'ordre ou de compatibilité ne canalisait son évolution [La66]. C'est alors qu'une deuxième école est venue prôner l'idée d'un langage universel dont l'ambition serait de

satisfaire tous les utilisateurs dans tous les domaines.PL/1 et Algol 68 en sont les résultats les plus caractéristiques, langages très vastes, aux multiples possibilités, mais où la recherche de la généralité s'est faite aux dépens de la présence de moyens "naturels" d'expression adaptés à des applications déterminées.

Devant cette multiplication du nombre des langages, que l'école "universaliste" n'a fait qu'encourager, s'est vite posé un problème technique, mais aussi économique, que venait aggraver l'apparition de types de machines de plus en plus divers. En effet, pour rendre disponibles ces langages, il fallait évidemment réaliser un nombre sans cesse croissant d'implémentations. On s'est d'abord contenté d'y travailler de façon plus ou moins artisanale puis, l'expérience aidant, la structure des compilateurs a pu être étudiée en profondeur et leur construction rendue plus systématique grâce à une nouvelle catégorie de langages spécialisés, les "compilateurs de compilateurs".

L'idée de départ est fort simple. Dans la définition de tout langage on distingue deux dimensions : la définition de sa syntaxe et la définition de sa sémantique. La syntaxe spécifie les conventions à respecter pour construire chaque phrase du langage, elle dégage la structure de chaque phrase correcte, et la sémantique attache aux constructions définies et structurées par la syntaxe un certain nombre d'actions qui constituent la "signification" de ces constructions. A ces deux dimensions correspondent deux types de travaux effectués par tout compilateur : un travail d'analyse et de structuration du programme source, qui constitue l'ossature du compilateur, et un travail de production du programme objet qui est déclenché lors de la reconnaissance de certaines structures syntaxiques par l'analyseur. Un compilateur de compilateurs a donc pour objet de faciliter l'écriture et le couplage entre elles de ces deux catégories d'activités. Dans la plupart des cas, la partie syntaxique est réalisée sous la forme d'un analyseur syntaxique général ou d'un générateur d'analyseurs auxquels on n'a qu'à donner une description de la grammaire du langage source. Ceci est tout à fait satisfaisant car il est fait une utilisation à peu près directe d'un formalisme qui peut rester très proche de celui employé pour la description syntaxique originale du langage. Par contre, les techniques qui permettent le couplage avec le travail de production et surtout la façon dont ce travail lui-même

doit être décrit sont loin d'alléger la tâche de l'écrivain de compilateurs dans la même mesure que cela a pu être fait pour le travail d'analyse syntaxique. En effet, le couplage entre analyse et production, et même, de façon plus généralé, entre actions syntaxiques et actions sémantiques, ne peut être accompli qu'en insérant dans le processus syntaxique des appels à des "routines" sémantiques. La position des points d'insertion est nécessairement dépendante de la méthode d'analyse syntaxique qu'impose le compilateur de compilateurs, et les routines elles-mêmes ne sont guère différentes de celles que l'on pouvait rencontrer dans les compilateurs "artisanaux". Tout au plus peut-on aider l'écrivain de compilateurs en mettant à sa disposition un répertoire de fonctions toutes prêtes pour consulter des tables, pour manipuler des piles, pour construire des listes, pour simplifier les problèmes lexicographiques, etc..., fonctions reconnues comme nécessaires à tout travail de compilation. Ainsi, les compilateurs de compilateurs ont conduit fort loin l'automatisation de l'analyse syntaxique, mais on comprend qu'ils aient déçu par l'absence de mécanismes du même ordre dans le domaine de la sémantique.

Simplifier, sinon supprimer, ces difficultés de compilation de nombreux langages, pouvoir en même temps bâtir des langages qui rassemblent les avantages des langages spécialisés et ceux des langages universels, tout en éliminant les inconvénients respectifs de ces deux approches, tels sont les objectifs que s'est fixé, vers 1966, une nouvelle ligne de travaux sur les langages de programmation. A l'origine, l'idée a été la suivante : puisque c'est le programmeur qui connait le mieux la nature des objets et des opérations dont il a besoin pour représenter le domaine de son application, il semble tout à fait normal de lui laisser le soin de les définir lui-même. Pour qu'il puisse y parvenir, il faut donc mettre à sa disposition des outils de définition qui n'exigent pas de lui les compétences d'un écrivain de compilateurs. Ces outils de définition faisant partie du langage lui-même, ce langage mérite alors le qualificatif "extensible".

C'est de là que sont issus les principes généraux qui ont guidé l'ensemble des travaux sur les langages extensibles. A partir d'une étude détaillée des éléments fondamentaux de la programmation, qui se profilent plus ou moins explicitement derrière la plupart des langages, il faut d'abord construire un "langage de base". Puis il faut concevoir des mécanismes,

dans la dimension syntaxique mais aussi, et surtout, dans la dimension sémantique, qui permettent à un utilisateur de combiner entre eux des éléments du langage de base, ou des éléments déjà plus élaborés, pour construire les éléments d'un langage "de plus haut niveau". On peut alors obtenir, de façon systématique et ordonnée, autant de langages que l'on veut, aussi spécialisés et "naturels" que le désire l'utilisateur, mais qui restent compatibles entre eux grâce au tronc commun du langage de base et des mécanismes qui ont servi à les définir.

Cependant, les travaux qui ont déjà été accomplis sur les langages extensibles, et encore plus ceux qui sont actuellement en cours, se sont tous plus ou moins détournés de l'un des objectifs originaux qui était de laisser à l'utilisateur "final" du langage le soin de définir lui-même toutes les caractéristiques syntaxiques et sémantiques de son langage. En effet, définir un langage revient toujours à construire, sous une forme particulière, une implémentation de ce langage. Cela requiert une expertise que n'a pas, en général, celui qui utilisera ce langage. Les mécanismes d'extensions dont se sert cet utilisateur doivent donc permettre de définir seulement certains aspects du langage qui ne remettent pas en cause la structure de son implémentation.

En revanche, il apparait de plus en plus que les mécanismes généraux de définition par extensions apportent aux écrivains de compilateurs des méthodes de descriptions syntaxiques et surtout sémantiques beaucoup plus complètes et bien mieux structurées que celles qui leur étaient proposées par les compilateurs de compilateurs. Cela est devenu encore plus évident depuis que se sont rejoints les travaux sur les langages extensibles et ceux sur les méthodes de définition formelle des langages de programmation. En effet, bien que poursuivant un but distinct de celui des langages extensibles, parce que se situant sur un plan beaucoup plus théorique, les travaux sur la définition formelle des langages de programmation s'inscrivent quand même dans le même courant : il s'est agi, dès le début, d'établir des bases formelles cohérentes à partir desquelles on pourrait exprimer "tout ce qu'il y a" dans un langage, c'est-à-dire définir sa syntaxe et sa sémantique. En fait, dès 1960, de nombreux travaux avaient déjà été entrepris et menés à bien sur le problème de la syntaxe, la plupart d'entre eux étant surtout consacrés à l'étude des propriétés des langages hors-

contexte et à la construction d'analyseurs pour ces langages. Mais ce n'est que vers 1966 - après une vue prophétique de John Mac Carthy en 1962 [Mc62a] - que le problème de la sémantique a vraiment été abordé et plusieurs techniques ont été développées. L'une d'entre elles, due au Laboratoire IBM de Vienne, a donné à la définition d'un langage une structure d'ensemble (syntaxe concrète, traducteur, syntaxe abstraite, interpréteur) qui rappelle celle d'une implémentation réelle : c'est ce qui, par la suite, a amorcé le rapprochement entre les méthodes de définition formelle et les techniques des langages extensibles. En effet, les travaux actuels sur l'extensibilité se sont orientés vers la recherche d'une méthode de définition formelle, qui non seulement représente de façon réaliste la structure d'une implémentation, mais qui permette aussi de traiter la définition formelle d'un langage pour en extraire automatiquement une implémentation de ce langage : c'est bien là le coeur du problème des langages extensibles.

Les travaux qui sont présentés dans cet ouvrage retracent ma participation à l'évolution des activités de recherche sur les langages extensibles. Ils s'insèrent dans une période allant de fin 1967 à aujourd'hui.

C'est en réalisant, en 1965, à l'Institut de Mathématiques Appliquées de Grenoble, un compilateur pour NLT, langage spécialisé dans le dépouillement d'enquêtes et l'extraction d'informations à partir de fichiers séquentiels, que j'ai découvert "sur le tas" les problèmes posés par la conception des langages et par leur compilation : construire un compilateur pour un langage déjà complexe, sans autre outil qu'un assembleur est un travail long et difficile. Cette expérience m'a incité à étudier de façon plus générale les problèmes de la compilation et je me suis alors intégré à l'équipe qui travaillait sur un compilateur de compilateurs, avec Jacques Cohen, Jean-Claude Boussard et Laurent Trilling. Dans ce cadre, j'ai travaillé plus particulièrement à la conception d'un langage, dit "Langage d'Edition", destiné à faciliter l'organisation et l'écriture de la phase pré-syntaxique des compilateurs. Une première version de ce langage a été décrite dans [Bo66], puis son intégration dans l'ensemble d'un compilateur de compilateurs présentée dans [Bo67]. J'ai réalisé une implémentation de ce langage d'édition sur matériel IBM 7044 après en avoir établi une définition précise et complète [Jo67a, Co67].

C'est au cours de ce travail que j'ai pu me rendre compte de l'importance qu'il y a à rapprocher autant que possible les techniques d'implémentation et les méthodes plus ou moins formelles de définition. Définir formellement un langage de programmation n'est vraiment justifié que si la définition envisagée a une forme et un contenu utile de façon aussi directe que possible à ceux qui ont le plus besoin de connaître en profondeur ce qu'il y a dans ce langage, c'est-à-dire, avant tout, ceux qui veulent l'implémenter pour le rendre disponible à une communauté d'utilisateurs. S'inspirer d'une définition formelle pour réaliser une implémentation impose alors une certaine discipline pour la structuration des algorithmes d'un traducteur ou d'un interpréteur, mais encore faut-il que cette définition formelle donne des algorithmes en question une image raisonnablement proche de ce qui peut se programmer.

Les années 1965, 1966 et 1967 étaient aussi celles où Algol X, avant qu'il ne devienne Algol 68, commençait à faire parler de lui, et j'ai progressivement pris part à diverses activités s'y rapportant, en particulier au sein d'un groupe de travail de l'AFIRO. Dans le cadre des problèmes d'implémentation sur lesquels je travaillais, cela m'a conduit à chercher comment le formalisme de la définition syntaxique d'Algol X, qui a pris depuis lors le nom de W-grammaire, pouvait être utilisé aussi directement que possible par un compilateur. C'est dans cette intention que j'ai défini et programmé en LISP 1.5 [Jo67b] un certain nombre de transformations automatiques que l'on pourrait faire subir à un formalisme syntaxique de cette sorte pour que son utilisation par un analysuer puisse être envisagée. Mais ce travail m'a aussi permis de découvrir de façon très détaillée tout ce qu'il y avait dans Algol X, en particulier son mécanisme de modes et sa généralisation de la notion de conversion, dont la conception était toute nouvelle à cette époque.

C'est alors que j'ai été invité, en Novembre 1967, à participer aux travaux de recherche dirigés par T.E. Cheatham qui était, à cette époque, Président de "Massachusetts Computer Associates" en même temps que Professeur de "Computer Science" à l'Université d'Harvard. Le sujet qui m'était proposé était nouveau pour moi : il s'agissait de concevoir et d'implémenter un langage "extensible". Les travaux dans ce domaine étaient encore rares et il a d'abord fallu déterminer la façon dont nous

allions procéder pour définir un tel langage. C'est ainsi qu'est devenue explicite, probablement pour la première fois avec ELF (Extensible Language Facility [Ch68]), l'architecture générale d'un langage extensible : il y a un langage de base sur lequel viennent se "brancher" des mécanismes d'extension pour les deux dimensions de la définition d'un langage, la syntaxe et la sémantique. Comme T.E. Cheatham avait déjà conçu des mécanismes syntaxiques [Ch66] qui pourraient être repris ultérieurement pour le langage extensible que nous allions définir, il a été décidé de consacrer l'essentiel des activités aux problèmes du langage de base et de la sémantique. C'est ainsi que j'ai pris en charge la coordination technique des travaux du groupe "Langages Extensibles" de Computer Associates, dont faisaient partie A. Fischer, S.A. Schuman, et auxquels se sont joints ensuite M.M. Hammer, puis P. Moskovites et D. Baeristo lorsqu'il s'est agi de réaliser pratiquement une implémentation. Après environ un an et demi de travail, le langage BASEL était défini [Fi68, Jo68, Jo69b, Jo70a, Jo70b] et nous avions l'assurance qu'il était construit de façon cohérente et qu'il était implémentable sans trop de difficulté, bien qu'étant proche parent d'Algol 68 en raison de mon intérêt d'alors pour l'effort de conception de ce langage [Jo69a]. Il était possible d'affirmer cela grâce à l'existence de la définition formelle de BASEL que, dès le début, j'avais entrepris de construire progressivement pour contrôler l'évolution et la cohérence des travaux du groupe [Ha69, Jo60c, Jo69d], et nous avons eu confirmation du bien fondé de notre certitude, car huit mois ont suffi à trois personnes du groupe pour réaliser une implémentation de BASEL calquée sur le fonctionnement de sa définition formelle.

Lors de mon retour en France, en Janvier 1970, dans le cadre du Centre Scientifique IBM de Grenoble, j'ai entrepris de poursuivre l'étude du problème des langages extensibles, estimant que c'était là, en particulier pour un constructeur - et vendeur - de matériel informatique, une façon réaliste d'envisager l'avenir des langages de programmation si l'on considère la variété sans cesse grandissante des communautés d'utilisateurs et des applications qui les intéressent [Jo70c]. En partant de l'expérience acquise avec la conception de BASEL dont, à Grenoble, Madame Bajar a étudié en détail dans sa thèse [Ba73] le mécanisme de modes, et en essayant toujours de rendre plus clair les liens qu'il y

a entre définition formelle, mécanismes d'extension et techniques d'implémentation [Jo72b], je me suis surtout attaché à l'étude de l'extensibilité dans le domaine de la sémantique : comment définir de nouvelles opérations dans un langage, autrement qu'avec des procédures classiques ? Comment rendre un langage capable de construire et de manipuler correctement de nouvelles sortes d'objets, autrement que par les mécanismes finalement assez rigides des modes d'Algol 68 ou de BASEL [Bo71, Jo73] ? Comment envisager, si cela a une signification, la possibilité de définir de nouvelles formes de contrôle de l'exécution dans un langage ? Pour tout cela quelle serait la forme souhaitable d'un langage de base ?

Après une première ébauche, fondée sur le concept de mode classique, à laquelle S. Schuman a joint un mécanisme très évolué d'extensibilité syntaxique [Sc70] qui a été repris et implémenté par Ph. Chatelin [Cw72], puis utilisé par B. Willis [Wl74], P. Cousot [Cs73], D. Bert [Be73, Be74], et M. Cabric [Ca74], j'ai cherché, avec D. Bert à établir des bases de départ différentes de celles de langages comme BASEL afin de vraiment rendre possible la construction, au moyen de simples déclarations, de la sémantique d'un langage. Le but était donc d'abord de parvenir à une analyse des problèmes de la sémantique qui permette de concevoir un système de déclarations au moins aussi évolué que ceux que l'on a déjà pu proposer pour la syntaxe. C'est seulement alors que l'on aura atteint l'objectif des langages extensibles : pouvoir définir tous les aspects d'un langage en déclarant simplement la forme et la signification des élements de ce langage, de façon à l'adapter au mieux à une application déterminée.

Cet objectif des langages extensibles, nous avons cherché à le préciser, et nous avons aussi cherché à déterminer quelques-uns des moyens pour l'atteindre lors du Symposium International sur les Langages Extensibles organisé en Septembre 1971 à Saint-Pierre-de-Chartreuse par S. Schuman, M. Griffiths et moi-même [Sc71a]. Mais, même à cette époque, tout n'était pas encore bien clairement délimité sur ce sujet. Depuis lors, avec le renfort des travaux sur les méthodes de définition formelle, les directions dans lesquelles il convenait de travailler se sont peu à peu précisées. D. Bert a pu ainsi proposer dans sa thèse [Be73] un langage de base mettant en évidence certains éléments primitifs du contrôle de l'exécution

et divers moyens de les combiner pour obtenir des formes de contrôle évoluées : parallélisme, coroutines, etc.... Dans le même temps, en rapprochant le concept de mode de quelques notions très rudimentaires sur les ensembles, j'ai construit un mécanisme permettant de définir par déclarations un aspect central de la sémantique statique d'un langage, qui concerne la vérification statique des modes, l'appel automatique des conversions, la génération de vérification dynamique, etc... [Jo71, Jo72]. D. Bert a utilisé quelques-uns des points de départ de ce mécanisme dans sa thèse, et j'estime que ce travail sur la sémantique statique d'un langage doit être poursuivi, ainsi qu'une réflexion plus approfondie sur le contenu d'un langage de base qui entrave le moins possible la liberté de définir par extension la sémantique d'un autre langage. C'est d'ailleurs œ que j'ai proposé [Jo74] au groupe de travail WG2.1 de l'IFIP comme point de départ dans une exploration de voies nouvelles pour la conception de langages de programmation, maintenant que, dans ce groupe, nous avons terminé [vW69] - et même révisé [vW74] - la définition d'Algol 68. Construire un formalisme qui rende la sémantique - au moins statique - aussi aisément abordable que l'est déjà la syntaxe est sans aucun doute le pas à franchir pour faire des langages extensibles des outils pratiques, afin de mettre à la disposition d'utilisateurs très divers des langages de programmation correctement adaptés à leurs problèmes.

Dans les chapitres qui suivent, je fais d'abord, au chapitre 2, une présentation générale de l'organisation d'un langage extensible telle que je l'envisage. Puis, au chapitre 3, je fais une étude détaillée de la façon dont j'ai traité le problème du langage de base, d'abord avec BASEL et sa définition formelle, puis avec L0, un langage dont je donne une description rapide afin de suggérer une orientation souhaitable pour la conception d'un langage de base.

Ensuite, au chapitre 4, j'étudie le problème essentiel des langages extensibles aujourd'hui, c'est-à-dire celui de l'extensibilité sémantique, en me limitant au domaine de la sémantique statique : après une étude critique de mécanismes déjà classiques, je propose une amélioration très importante du mécanisme des modes, puis je présente le mécanisme des classes qui, à partir de notions très élémentaires, permet vraiment d'aborder avec beaucoup de liberté l'extensibilité de la sémantique statique.

Ce chapitre se termine, lui aussi, sur des suggestions pour élargir encore le domaine d'application du mécanisme des classes.

Enfin, en conclusion, je rappelle brièvement les étapes à franchir pour réaliser un langage extensible, puis j'indique ce qui, à mon avis, doit être aujourd'hui l'orientation des travaux de recherche sur les langages, moyens d'accès à un ordinateur. Les langages extensibles ne sont en effet que l'une des composantes d'un courant beaucoup plus important.

2. ORGANISATION D'UN LANGAGE EXTENSIBLE

-:-

Les langages extensibles ont l'ambition, d'une part, de proposer une solution cohérente aux problèmes posés par l'absence d'organisation qu'implique la multiplicité de langages spécialisés conçus de façon indépendante, et, d'autre part, d'offrir bien plus de souplesse et de facilité d'expression que ne peut le faire le compromis des langages dits "universels". L'objectif des langages extensibles est donc de permettre la définition ordonnée et l'implémentation automatique de langages où, comme dans les langages spécialisés, l'utilisateur trouvera des moyens d'expression "naturels" qu'il pourra d'ailleurs, en général, définir lui-même.

Pour cela, un langage extensible est conçu comme étant un langage de programmation qui, en plus de la possibilité d'écrire des programmes d'application, fournit divers outils pour modifier sa propre définition et, en conséquence, sa propre implémentation.

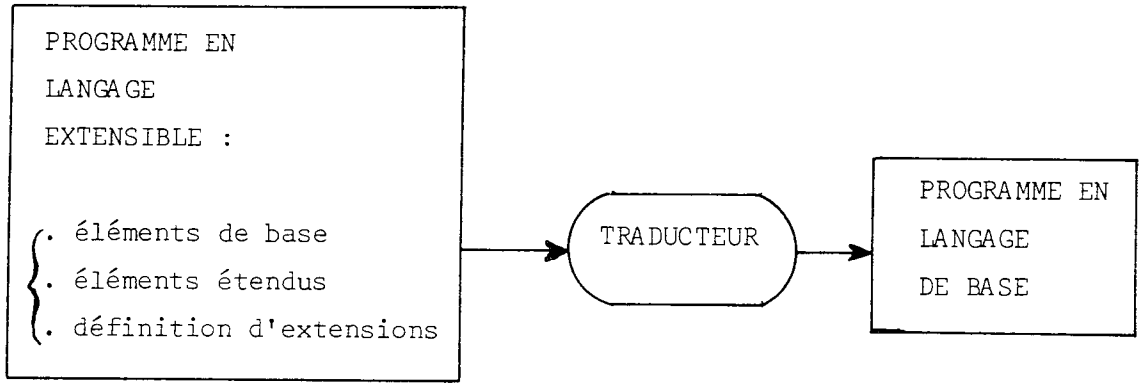
Comme, de façon maintenant classique, on distingue deux dimensions dans la définition d'un langage, la syntaxe et la sémantique, dans un langage extensible, il faut fournir des moyens pour modifier la syntaxe et des moyens pour modifier la sémantique, de façon à adopter le langage à l'utilisation envisagée : ces moyens sont appelés mécanismes d'extension. Mais il est également nécessaire de prévoir un "point de départ" pour les extensions : ce point de départ est présent dans un langage extensible sous la forme d'un langage de base. Le langage de base est un langage de programmation qui contient un inventaire d'éléments "primitifs" de la programmation, et les mécanismes d'extension permettent de combiner entre eux ces éléments, ou des éléments déjà construits, pour en construire de nouveaux. Lorsqu'un élément construit est utilisé dans le langage étendu dont il fait partie, son implémentation, établie automatiquement lors de sa définition, a pour rôle essentiel de le "ramener" à la combinaison d'éléments de base qui ont servi à le construire, en passant éventuellement par d'autres éléments construits qui auraient pu être employés dans sa définition. Le langage de base est donc la base, à la fois syntaxique et sémantique, de tout langage défini par extension.

Dans la suite de ce chapitre, j'indique d'abord brièvement quels sont, à partir de cette vue générale, les avantages pratiques immédiats que l'on peut attendre d'une telle approche, puis je décris la façon dont j'envisage l'organisation interne d'un langage extensible, en utilisant la structure d'une définition formelle interprétative, d'une part pour isoler clairement les rôles respectifs du langage de base et des divers mécanismes d'extension, d'autre part pour en déduire la forme souhaitable de l'implémentation d'un langage extensible.

2.1. VUE GENERALE DE L'EXTENSIBILITE

Comme on vient de le voir, un langage extensible est constitué d'un langage de base et de mécanismes d'extension. En général, un programme écrit dans un langage extensible utilisera donc des éléments primitifs du langage de base, des éléments définis par extensions et il effectuera lui aussi un certain nombre d'extension pour son usage propre.

Un tel programme est alors soumis à un processus de traduction qui a pour but de produire un autre programme, "sémantiquement équivalent", entièrement exprimé à l'aide d'éléments primitifs du langage de base :



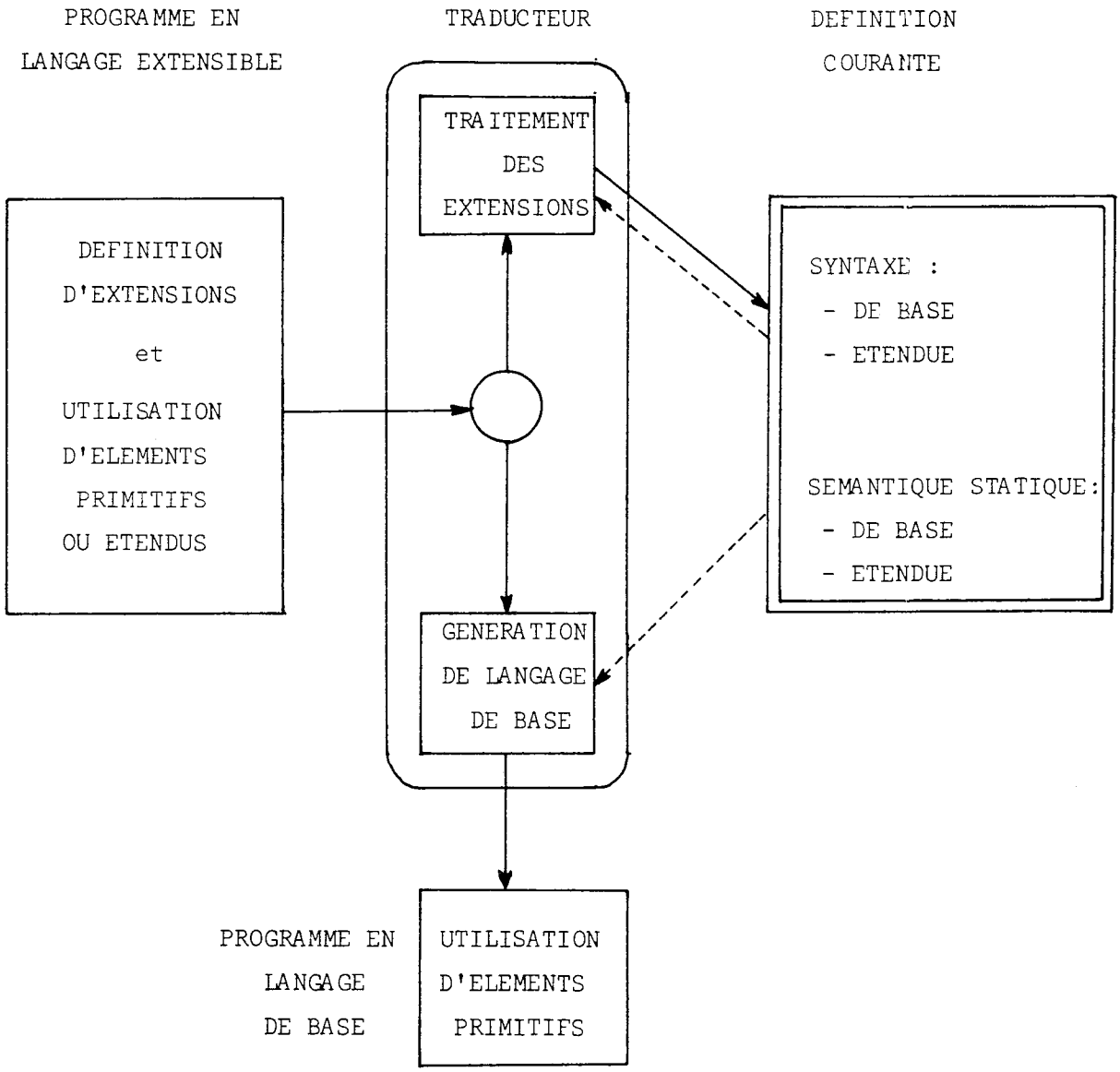
Le traducteur doit donc, d'une part, accepter ce qui fait partie du langage de base et transformer toute utilisation de ce qui a été défini par extension, d'autre part prendre en compte toute définition de nouvelle extension.

On peut ainsi, schématiquement, distinguer deux parties dans le traducteur :

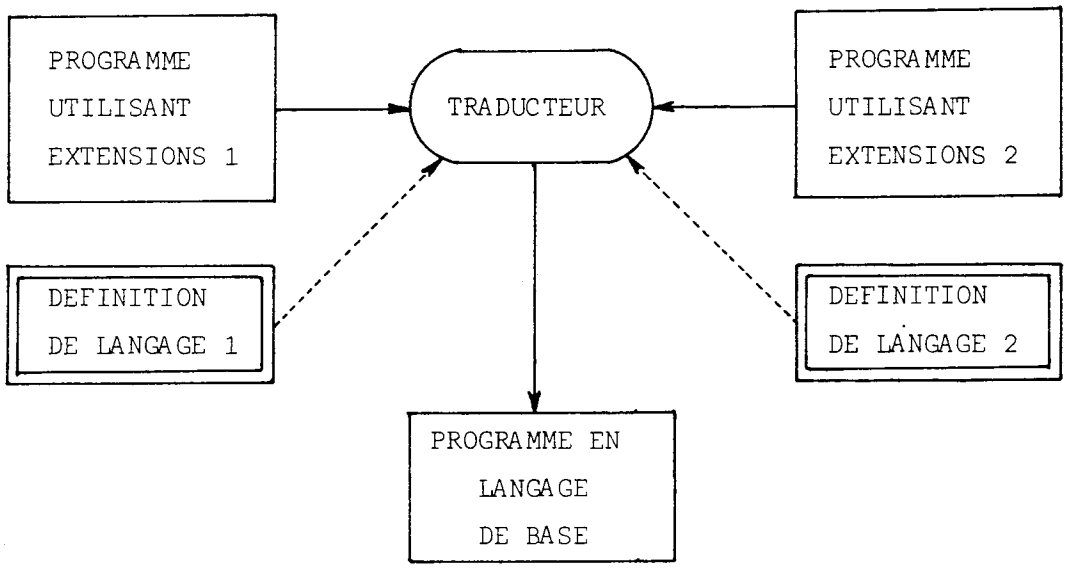
- une partie qui traite l'introduction de nouvelles extensions et dont l'effet est de modifier la définition courante du langage ;
- une partie qui effectue le travail de traduction proprement dit et qui, pour cela, a nécessairement connaissance de l'état courant de la définition du langage.

On voit alors apparaître le rôle essentiel d'une représentation de la définition du langage, utilisée par le traducteur et mise à jour par les extensions. Cette représentation de la définition ne concerne que ce qui est effectivement intéressant pour le traducteur, c'est-à-dire la syntaxe et la sémantique statique. Elle est "initialisée" avec une description de la syntaxe et de la sémantique statique du langage de base et elle évolue lors de l'utilisation des mécanismes d'extension.

Un schéma plus détaillé que le précédent serait donc :

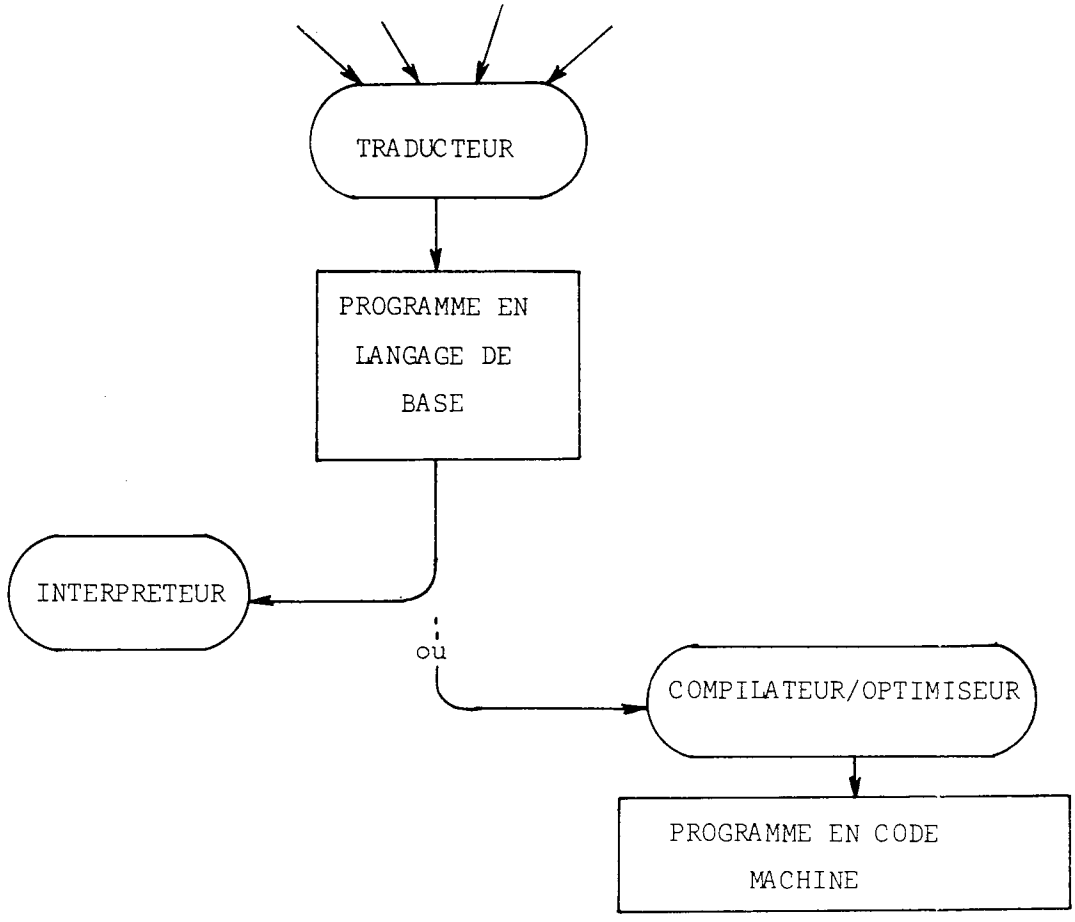


Ce fonctionnement général fait que tout langage défini par extension possède, à chaque instant, une définition organisée de façon cohérente et un traducteur opérationnel. Un tel outil simplifie donc le travail de conception des langages de programmation : on peut définir autant de langages spécialisés que l'on veut pour des domaines d'applications variés, en ayant automatiquement leurs implémentations. Tous les langages définis avec un même langage extensible ayant une même organisation de leur définition, un même mécanisme de traduction, et un même langage de base, les programmes écrits en différents langages étendus sont compatibles :



Ceci supprime l'un des inconvénients majeurs des langages spécialisés conventionnels, qui est leur incompatibilité mutuelle. En effet, jamais deux d'entre eux n'ont été conçus pour faire mutuellement appel l'un à l'autre dans des programmes. D'où une difficulté pour les utilisateurs mis en face de problèmes arbitrairement rendus "hybrides" par cette conception cloisonnée des langages.

Enfin, étant donné cette architecture générale d'un langage extensible, il résulte que toute la gamme des langages étendus se satisfait d'un processus unique d'interprétation ou de compilation, qui concerne le langage de base :



En particulier, si le compilateur du langage de base comprend un optimiseur, tous les programmes écrits dans tous les langages étendus en bénéficieront.

Ainsi, à partir de ce principe général de fonctionnement, on peut prévoir que les langages extensibles vont apporter une amélioration importante dans la facilité avec laquelle les programmeurs pourront exprimer leurs algorithmes. On peut d'ailleurs estimer que les langages extensibles regroupent les principaux avantages des langages classiques - spécialisés et universels - tout en éliminant leurs inconvénients les plus sérieux.

2.2. DEFINITION FORMELLE ET EXTENSIBILITE

Utiliser des mécanismes d'extension a pour effet d'apporter des modifications à la définition courante du langage, modifications qui sont aussitôt prises en compte par l'implémentation. Il est donc essentiel d'aborder l'architecture d'un langage extensible en cherchant à rendre aussi clairs que possible les liens qu'il y a entre la définition formelle du langage et son implémentation. C'est ce qui va permettre de situer de façon précise le rôle du langage de base et les places respectives des mécanismes d'extension syntaxique et sémantique, d'abord à l'intérieur de l'organisation d'une définition formelle, puis dans l'implémentation d'un langage extensible.

2.2.1. SCHEMA D'UNE DEFINITION FORMELLE

Présentées par Wegner dans [Wn69], et aussi par Dijkstra dans [Di72], plusieurs méthodes ont été proposées pour définir formellement un langage de programmation. Dans sa thèse [Vi74], Vidart les rappelle brièvement :

- la méthode interprétative,
- la méthode fonctionnelle,
- la méthode axiomatique.

Sans entrer ici dans une description de chacune de ces méthodes, il est aujourd'hui reconnu qu'avec l'organisation actuelle du fonctionnement des machines, c'est la méthode interprétative qui permet de représenter de la façon la plus fidèle l'architecture réelle de l'implémentation d'un langage.

En quelques mots, le principe de cette méthode est le suivant :

- Soit une machine ("abstraite") M définie par :
  - . un répertoire R d'opérations, dont l'utilisation obéit à un certain nombre de règles ;
  - . un interpréteur I, "force motrice" de M, qui pour chaque opération de R effectue une transformation dans un environnement E.
    Pour M, un programme est donc un ensemble d'appels à des opérations de R écrits selon les règles prescrites, et l'exécution de ce programme est une suite de transformations effectuées par I dans E.
- Soit L le langage dont on veut construire une définition.

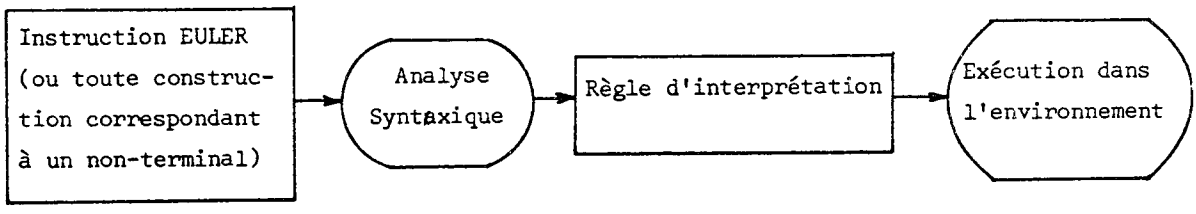
Tout mécanisme qui à tout programme P écrit en L fait correspondre un ensemble d'appels à des opérations exécutables par M est une définition interprétative de L par M.

La méthode interprétative a souvent été employée pour définir des langages : LISP [Mc62], EULER [Wi66], PL1 [Al68a , Al68b, Fl68, Wa68], Algol 68, surtout dans sa version révisée [vW74], et aussi BASEL [Ha69, Jo69c, Jo69d] ont des définitions construites sur ce modèle. D'autre part, la méthode dite "de Vienne", utilisée d'abord pour définir PL/1, mais aussi pour une définition d'Algol 60 d'une concision remarquable [Lr68], est l'un des formalismes les plus connus pour construire de telles définitions [Lu68, Lu69, Ne71, Wn73].

Dans l'absolu, une telle forme de définition n'apprend évidemment pas tout sur le langage L qu'elle est censée définir. On ne peut en effet considérer que L est complètement défini que si la machine M à son tour est définie. En fait, dans la pratique, la machine M est choisie d'une nature telle que sa définition peut être donnée sans difficulté dans une langue naturelle, parfois avec l'aide de quelques notations mathématiques : son répertoire R d'opération est "petit" et l'organisation de l'environnement E sur lequel travaille l'interpréteur est "simple". Mais l'on peut fort bien imaginer toute une "cascade" de définitions interprétatives aboutissant pour terminer à une machine primitive qui ne serait guère plus évoluée qu'une machine de Turing. Toutefois, pour le problème abordé ici, ce n'est pas ce souci d'avoir une définition explicitement construite sur des bases formelles classiques qui est retenu, mais plutôt celui d'arriver à une définition interprétative organisée de telle façon qu'elle puisse constituer le noyau d'une implémentation du langage défini. (Cependant, ces deux points de vue ne sont pas du tout incompatibles).

Dans ce qui suit on cherche donc d'abord à déterminer les divers composants d'une définition et leur organisation globale afin de pouvoir y retrouver les divers traitements qu'effectue l'implémentation d'un langage.

Une première façon très simple d'envisager l'organisation d'une définition interprétative est illustrée par la définition d'EULER [Wi66]. Schématiquement, la structure syntaxique d'EULER est définie par des règles de grammaire hors-contexte et sa sémantique par l'effet que l'exécution d'une séquence de règles d'interprétation produit sur un ensemble de variables appelées l'environnement. Il existe, pour EULER, une correspondance unique entre les règles syntaxiques et les règles d'interprétation, et la séquence des règles d'interprétation qui est exécutée est entièrement déterminée par la suite des règles syntaxiques qui sont appliquées pendant une analyse syntaxique réductive d'un programme :



Les règles d'interprétation sont exprimées de façon algorithmique à l'aide d'une notation qui constitue en fait, à son tour, un petit langage de programmation à syntaxe et sémantique prédéfinies.

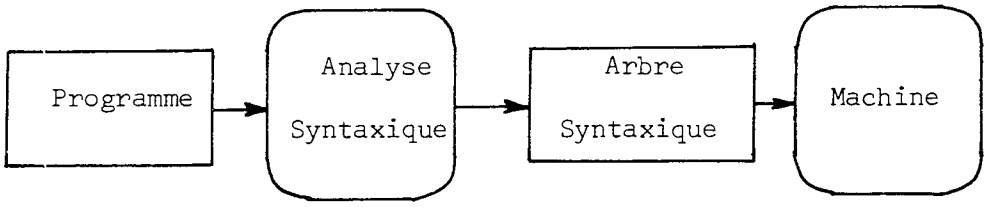
Ainsi, dans la définition d'EULER, la machine M dont il était question plus haut est constituée par :

- un répertoire R d'opérations qui entrent dans le langage primitif des règles d'interprétation,
- un interpréteur (implicite) très simple, capable de réaliser dans l'environnement l'effet de chaque opération de R.

Cette machine étant donnée, la définition d'EULER proprement dite est donc constituée du couplage entre règles syntaxiques et règles d'interprétation.

Cependant, cette forme de définition a le grave inconvénient de ne pas rendre explicite la distinction entre ce qui, dans un programme, peut être traité de façon statique et ce qui doit l'être de façon dynamique. En effet, le déclenchement de l'exécution d'opérations de la machine M est directement commandé par l'analyse syntaxique. Il en résulte un certain nombre de restrictions assez sévères sur la forme de cette syntaxe, et une "rigidité" de l'ensemble difficilement compatible avec l'objectif visé dans la définition d'un langage extensible. En particulier, on voit très mal comment y introduire un moyen d'étendre la sémantique, c'est-à-dire de modifier ce qui est "en aval" de l'analyse syntaxique hors contexte.

On doit donc chercher à isoler, dans le traitement formel d'un programme, plusieurs étapes qui permettent de distinguer ce qui peut être statique de ce qui doit rester dynamique, à l'image d'une implémentation réelle. Dans ce sens, une première amélioration possible, à partir d'une définition organisée comme celle d'EULER, est d'effectuer un premier "passage" qui construise l'arbre d'analyse syntaxique d'un programme, puis de confier cet arbre à une machine qui en effectue un parcours et exécute pour chaque noeud rencontré les mêmes règles d'interprétation que celles qui étaient associées à la règle de grammaire qui a servi à construire ce noeud, avec le même environnement que la machine précédente.

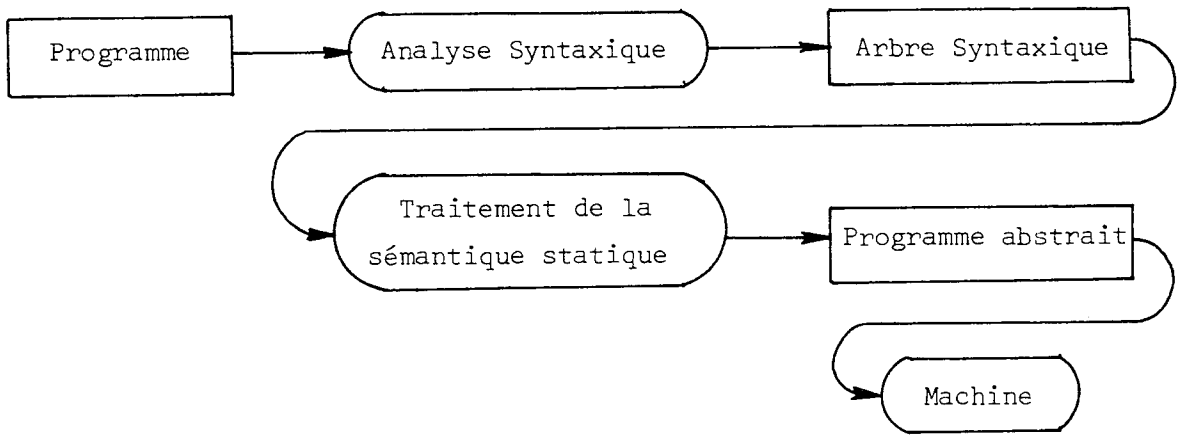


Ceci isole donc bien tout ce qui peut être décrit au moins par une grammaire hors contexte.

Cependant, en général, les caractéristiques d'un langage permettent aussi de rendre statique une partie du traitement qui se situe après l'analyse syntaxique. C'est le cas, en particulier, de l'association déclaration-utilisation et de tout ce qui concerne les vérifications de types et la génération automatique d'appels à des conversions ou à des vérifications dynamiques. A la suite de Griffiths [Gr73], ces caractéristiques d'un langage, qui ne sont plus exprimables par des règles de grammaire hors-contexte, mais qui peuvent être retirées du traitement dynamique sans changer la signification d'un programme, sont regroupées sous la dénomination de "sémantique statique".

En partant de l'arbre d'analyse d'un programme, le traitement correspondant à la sémantique statique construit alors une autre représentation du programme qui, comme dans le schéma de Vienne, peut être débarassé des contraintes syntaxiques originales, mais qui, surtout, ignore toute information explicite sur la notion de type, contient les appels aux conversions et établit lui-même le lien entre déclaration et utilisation. Cette représentation, qu'on a pris l'habitude d'appeler programme abstrait, est ensuite confiée à une machine qui prend en charge ce qui reste à définir, c'est-à-dire la sémantique dynamique. C'est donc le répertoire d'opérations de cette machine qui sert pour construire le programme abstrait, et cette machine fonctionne selon un principe analogue aux précédentes, avec interpréteur et environnement.

Mais, maintenant, la machine M peut être plus simple que lorsqu'elle avait aussi à tenir compte de tous les problèmes que l'on vient de reporter sur la sémantique statique :



Un tel schéma pour une définition, image réaliste de l'architecture d'une implémentation, permet alors de situer clairement les divers composants d'un langage extensible et de définir leurs rôles respectifs.

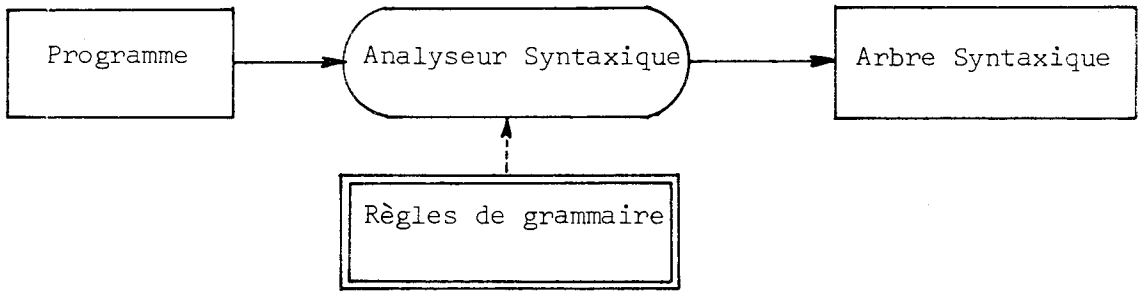
2.2.2. INTRODUCTION DE L'EXTENSIBILITE DANS UNE DEFINITION

A partir des considérations précédentes sur la forme d'une définition, on constate, comme cela avait été dit au début de ce chapitre, que des extensions peuvent être effectuées dans le domaine de la syntaxe et dans celui de la sémantique. Cependant, on peut maintenant être plus précis et parler de :

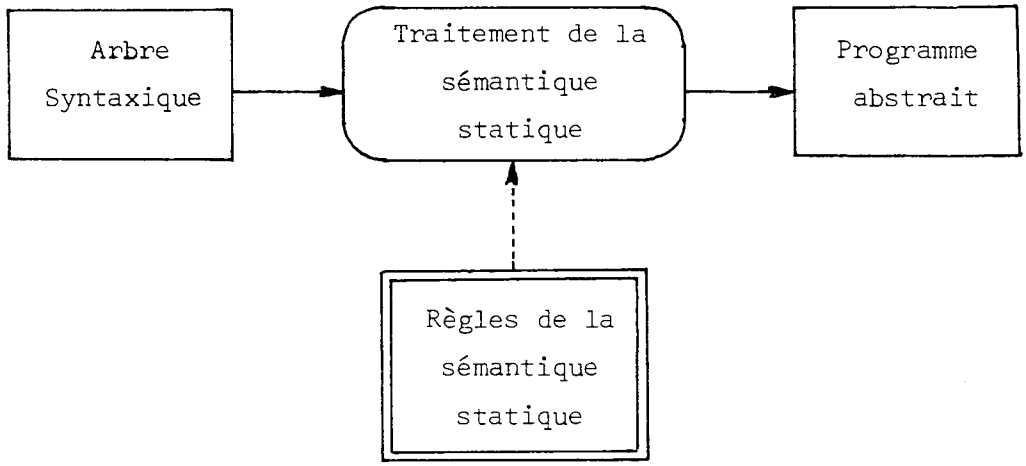
- l'extensibilité de la syntaxe ;
- l'extensibilité de la sémantique statique ;
- l'extensibilité de la sémantique dynamique.

Afin d'établir quelle est la place de chaque mécanisme d'extension et de décrire son rôle, on commence par introduire très brièvement un modèle du fonctionnement de chaque composant de la définition interprétative.

En ce qui concerne l'analyse syntaxique, on prendra le schéma le plus simple pour la suite, c'est-à-dire celui d'un analyseur qui utilise directement une représentation des règles de grammaire pour construire l'arbre syntaxique d'un programme :



De même, pour la sémantique statique, le traitement utilisera une représentation des règles qui définissent cette sémantique statique. (Au chapitre 4, on étudie de telles règles pour toutes les questions liées à la notion de type).



Enfin, pour la sémantique dynamique, on donne de la machine M l'image (simplificatrice) suivante :

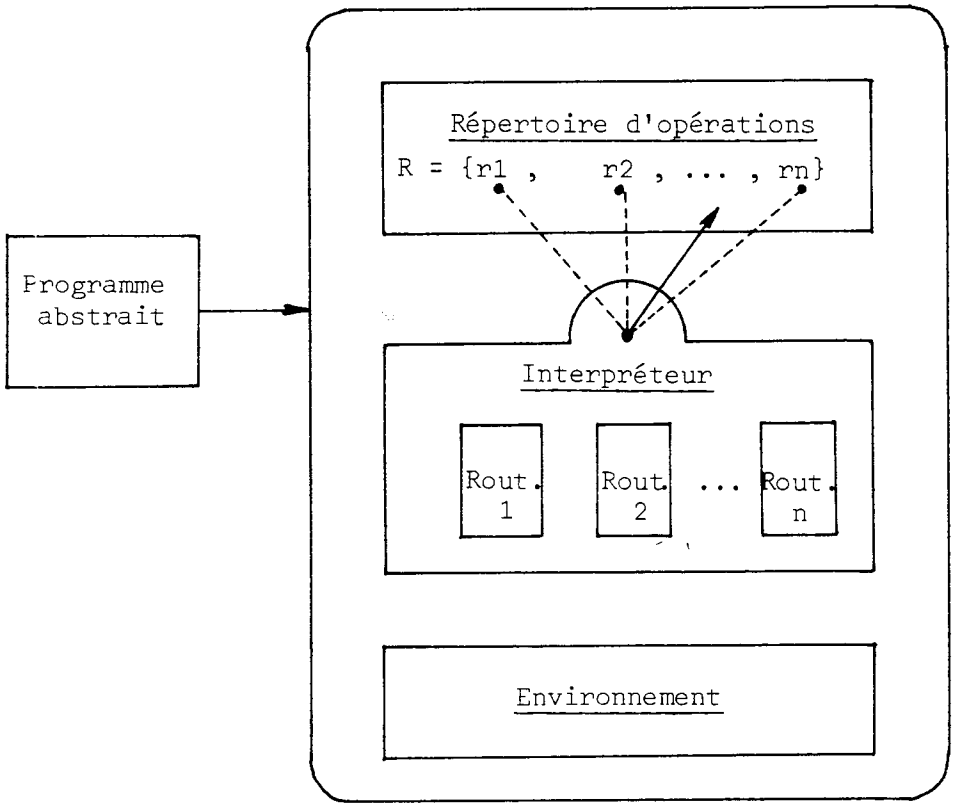
- R = {r1, r2, ..., rn} est le répertoire d'opérations, où ri ($1 \leq i \leq n$) est une opération dont l'utilisation a la forme :

  ri (a1, a2, ..., api)

  les aj ($1 \leq j \leq pi$) étant eux-mêmes des utilisations d'opérations de R, servant d'arguments à l'utilisation de ri.

- L'interpréteur peut être vu comme un aiguillage [Sc71] qui fonctionne de la façon suivante :

  . il fait une lecture du programme abstrait ;
  . au cours de cette lecture, l'utilisation d'une opération ri provoque l'évaluation de ses pi arguments, avec stockage temporaire de leurs résultats dans l'environnement (qui peut être vu, par exemple, comme une pile) ;
  . les arguments étant évalués, l'aiguillage prend la $i^{ème}$ position et exécute la "routine d'interprétation" associée à cette position dont le rôle est d'appeler les valeurs les arguments à partir de l'environnement, puis d'effectuer les actions propres à l'opération ri, et enfin de renvoyer un résultat éventuel dans l'environnement.



A partir de ces quelques remarques, on peut maintenant examiner comment les composants d'un langage extensible entrent dans une définition interprétative, et par suite dans une implémentation, en commençant par le langage de base.

2.2.2.1. Définition du langage de base

Le langage de base a une syntaxe, une sémantique statique et une sémantique dynamique prédéfinies.

Il y a donc :

- une grammaire Gb qui contient les règles syntaxiques du langage de base,
- un ensemble Sb de règles pour la sémantique statique du langage de base,
- un répertoire Rb d'opérations - avec les routines associées - pour la machine qui interprète les programmes abstraits obtenus à partir de programmes en langage de base.

Gb, Sb et Rb sont donc les informations initiales utilisées par les mécanismes de la définition - et de l'implémentation -.

2.2.2.2. Extensibilité de la syntaxe

Etendre la syntaxe a pour but de modifier l'aspect externe du langage, en y introduisant la possibilité d'utiliser des notations et d'écrire des "phrases" dont la forme et le style sont adaptés au domaine d'application auquel on destine le langage.

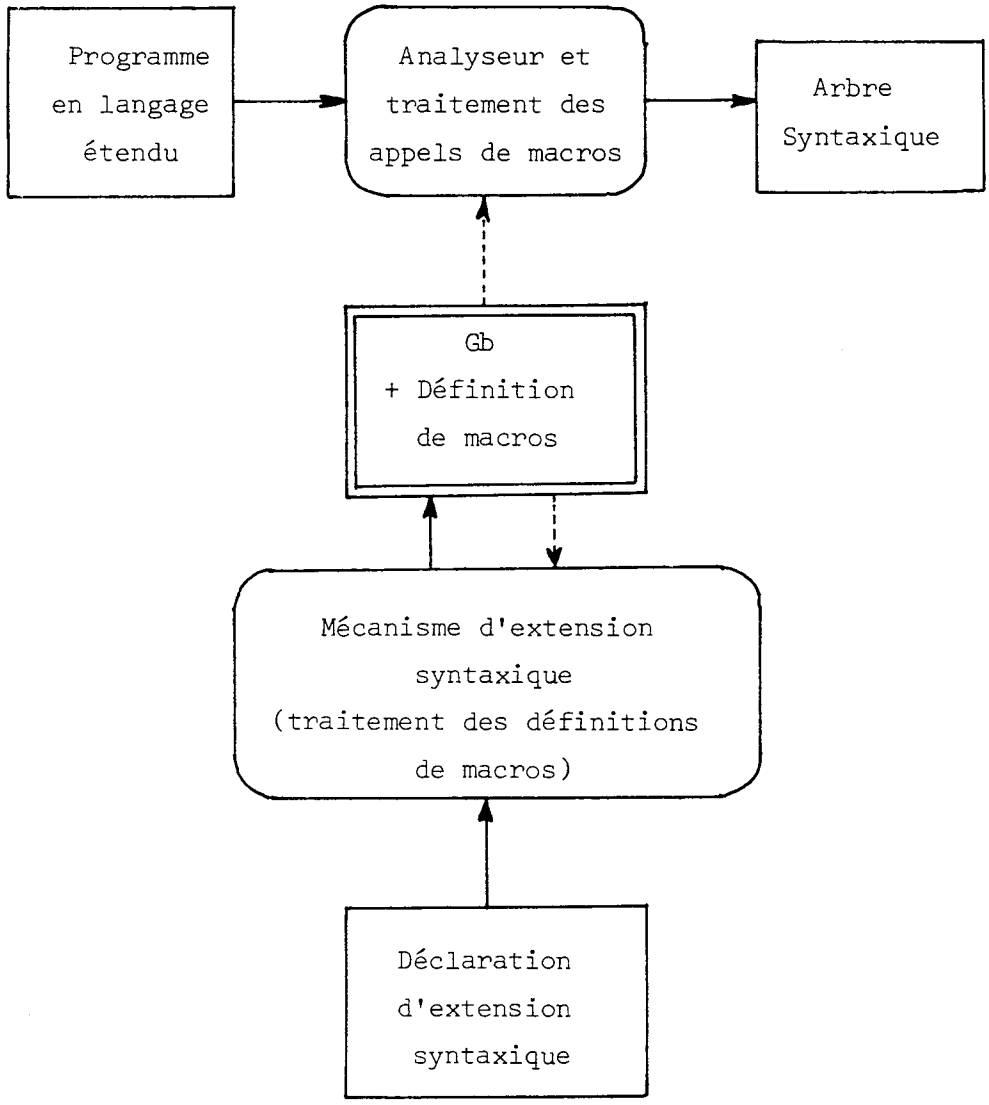
Un mécanisme d'extension syntaxique peut être rendu accessible par un système de déclarations qui permet d'exprimer :

- la forme de la nouvelle notation ou construction que l'on définit ;
- les endroits, dans le reste du langage, où cette nouvelle notation ou construction pourra être utilisée ;
- la structure de la traduction dans l'arbre syntaxique de cette nouvelle notation ou construction, c'est-à-dire sa "signification" exprimée à l'aide de constructions déjà définies.

D'une façon générale, dans le cadre d'une grammaire hors contexte, une extension syntaxique sera donc définie par l'introduction d'une nouvelle règle, avec :

- un membre droit qui définit la forme de la nouvelle construction ;
- un membre gauche, non-terminal, qui exprime les endroits où elle peut être utilisée ;
- un remplacement, éventuellement accompagné de conditions, qui construit la signification à l'aide du langage existant, et qui prend ses paramètres dans le membre droit de la règle.

Il s'agit donc d'un mécanisme de macros dirigé par la syntaxe. Pour introduire cela dans une définition interprétative, il faut que l'analyseur syntaxique qui, dans le modèle utilisé ici, se sert d'une description de la grammaire, sache reconnaître les règles qui provoquent un remplacement, ces règles étant rangées avec les autres (celles de Gb) par le mécanisme d'extension syntaxique. D'autre part, c'est l'analyseur lui-même qui est chargé d'effectuer le remplacement des utilisations de nouvelles constructions par leur signification :



De nombreux mécanismes d'extension syntaxique pouvant s'inscrire dans ce schéma ont déjà été proposés, du plus simple et rudimentaire, jusqu'au trop général.

Au niveau le plus élémentaire on trouve le mécanisme des opérateurs d'Algol 68, qui permet de définir de nouveaux opérateurs - monadiques ou dyadiques -, de choisir leur place dans la hiérarchie des priorités d'opérateurs et de définir leur remplacement par des appels de procédures à un ou deux paramètres. Dans ce cas, on peut donc considérer que l'analyseur utilise - parmi beaucoup d'autres informations - une table des priorités d'opérateurs, et que c'est cette table que modifie la déclaration d'un nouvel opérateur avec attribution d'une priorité. Dans une étude

sur l'extensibilité en Algol 68 [Jo73], j'ai donné un exemple d'utilisation de ce mécanisme pour définir une construction de la forme :

pourtout E en S telque P(E) effectuer A(E)

où S représente un ensemble, E un élément de S, P une propriété qu'a ou n'a pas un élément de S et A une opération quelconque sur un élément de S. Bien qu'il soit possible de définir des constructions de ce genre "à coups" d'opérateurs monadiques et dyadiques, on se rend vite compte que c'est au prix d'acrobaties sans rapport avec la simplicité de ce que l'on veut définir.

A l'autre extrême, on rencontre le mécanisme des macros syntaxiques présenté dans [Sc70], puis repris et implémenté par Chatelin [Cw72], qui donne libre accès à l'ensemble de la grammaire hors contexte du langage, et qui n'impose aucune restriction sur les propriétés du langage défini par cette grammaire. Ce système a été utilisé à plusieurs reprises pour des réalisations dans lesquelles on avait besoin de toute sa généralité. On retiendra en particulier l'implémentation des "Red-languages" de John Backus [Bk72] par P. Cousot [Cs73] et la production d'analyseurs pour langages définis par W-grammaires par D. Bert [Be74]. Cependant, le but du mécanisme est de servir à la traduction d'un langage étendu vers un langage de base. Les expériences faites à ce sujet avec les macros syntaxiques [Be73, Wl74, Ca74] ont provoqué une réaction unanime des utilisateurs : la généralité de ce système se paye par une inefficacité prohibitive (voir à ce sujet les remarques faites par M. Cabric dans la conclusion de son rapport de D.E.A. [Ca74]).

Entre ces deux extrêmes, plusieurs autres mécanismes ont été conçus. En particulier, on notera :

- le mécanisme des macros de Learenworth [Le66], qui limite l'ensemble des non-terminaux accessibles dans une définition de macro, et qui restreint sévèrement la forme des constructions que l'on peut définir.
- le mécanisme proposé par Vidart [Vi74] qui, dans un souci de simplicité et d'efficacité, utilise un modèle de compilation éprouvé ([Gr68, Gr74]), et effectue un prétraitement du remplacement de la définition d'une macro.

A la suite de tous ces efforts, il semble qu'un mécanisme satisfaisant puisse maintenant être mis au point en tenant compte de deux critères :

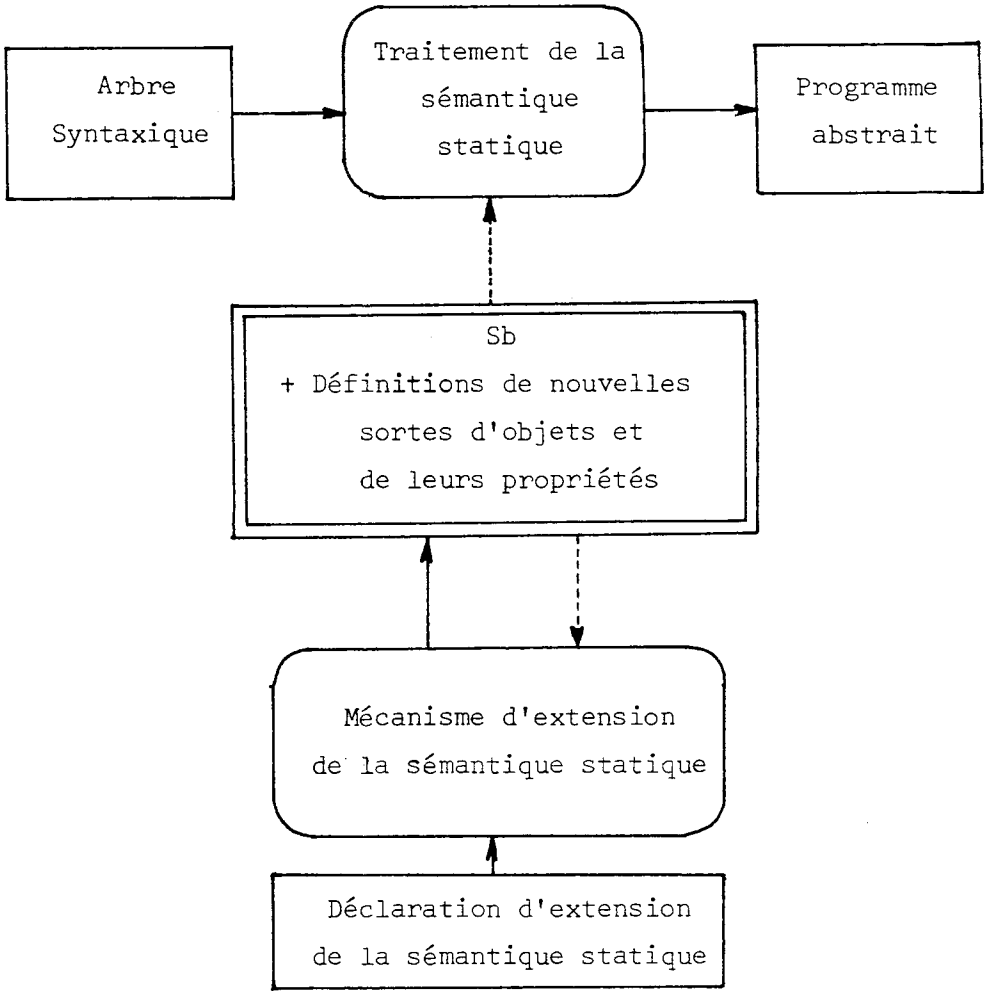
- l'efficacité, à obtenir en grande partie par un abandon d'une trop grande généralité recherchée jusqu'alors ;
- la simplicité d'utilisation, à rechercher dans une forme de déclaration qui n'exige pas de l'utilisateur une connaissance de l'ensemble de la grammaire du langage.

Dans le reste de cet ouvrage, le problème de l'extensibilité syntaxique ne sera plus abordé. En effet, l'importance et la quantité des travaux sur les mécanismes d'extension syntaxique a fait progresser les connaissances sur cette question à un point que l'on peut commencer à juger satisfaisant. Il n'en est pas encore de même - et de très loin - pour les questions portant sur la sémantique - aussi bien statique que dynamique. L'un des buts de cet ouvrage est donc d'apporter quelques lumières sur ce problème, et de suggérer diverses orientations de recherches qui semblent pouvoir conduire à des résultats intéressants et applicables, en particulier à propos de la sémantique du langage de base et à propos des mécanismes qui permettent d'envisager une véritable extensibilité de la sémantique statique.

2.2.2.3. Extensibilité de la sémantique statique

Etendre la sémantique statique permet, en particulier, de définir de nouvelles sortes d'objets manipulables dans le langage en spécifiant la configuration de ces objets et les règles de leur utilisation. Ceci est donc utile pour rendre disponible dans les langages, des objets - et leurs propriétés - qui donnent du domaine réel de l'application envisagée une image aussi directe et évidente que possible. C'est de cette façon que l'extensibilité de la sémantique statique est abordée en détail au chapitre 4.

Comme pour l'extensibilité de la syntaxe, le mécanisme d'extension de la sémantique statique est rendu accessible par un système de déclarations, qui sert à faire évoluer l'ensemble des règles de la sémantique statique :



2.2.2.4. Extensibilité de la sémantique dynamique

Tout ce qui est exprimé dans un langage défini par extension est construit sur la sémantique du langage de base, c'est-à-dire, finalement, avec les opérations du répertoire Rb.

Etendre la sémantique dynamique a pour objet de rajouter de nouvelles opérations à ce répertoire, en construisant aussi la routine d'interprétation associée. De telles extensions peuvent être considérées de deux façons différentes :

- D'une part, on peut vouloir rajouter une nouvelle opération au répertoire, en construisant "à la main" la routine associée et en donnant accès à cette opération à travers une nouvelle expression du langage

de base, elle aussi rajoutée "à la main". Il s'agit donc de "véritables" extensions de la sémantique dynamique, mais qui semblent en dehors de ce que permet normalement un langage extensible, car elles supposent d'intervenir directement au niveau du fonctionnement interne de l'implémentation.

- D'autre part, après avoir construit une procédure, ou une combinaison quelconque d'opérations (définissant par exemple une conversion automatique ou des propriétés attachées à un type d'objet défini par extension de la sémantique statique), on peut vouloir considérer qu'il s'agit d'une opération qui devrait être mise au "même niveau" que les opérations de Rb, c'est-à-dire reconnue et exécutée directement par l'interpréteur. Il s'agit alors du mécanisme de l'interpréteur extensible qui, de façon stricte, n'étend pas vraiment la sémantique dynamique, mais qui permet d'aborder l'implémentation d'un langage extensible en éliminant le risque d'inefficacité lié aux principes mêmes de la définition par extensions, comme cela a été décrit dans [Sc71] et [Jo72a]. Cette extensibilité de la sémantique dynamique est donc plus une technique d'implémentation qu'un mécanisme d'extension proprement dit. Mais, à mon avis, les travaux sur cette question seront déterminants pour rendre les langages extensibles pratiquement utilisables.

3. LE LANGAGE DE BASE

-:-:-:-:-:-:-:-:-:-:-

Comme on l'a vu au chapitre précédent, un langage extensible doit permettre de définir autant de langages "étendus" que l'on veut, tous construits à partir d'objets élémentaires disponibles dans un même langage de base.

Le rôle d'un langage de base est donc déterminant dans ce qu'un langage extensible permet de faire. En effet, tout ce qui est exprimé dans un langage étendu ne peut l'être que s'il existe une combinaison correspondante de constructions primitives du langage de base, car tout, finalement est ramené au langage de base par le principe même des mécanismes d'extension.

Dans ce chapitre, deux approches sont proposées. Sur la première, qui est celle du langage BASEL, j'ai travaillé aux Etats-Unis de novembre 1967 jusqu'en décembre 1969, dans le cadre d'un projet de conception d'un langage extensible. Ce chapitre en donne une description complète et présente la méthode que j'ai utilisée pour construire la définition formelle de BASEL, méthode qui peut s'appliquer à la plupart des langages et même des systèmes. Cet exposé sur BASEL se termine par quelques remarques générales sur ce langage et par une critique de sa qualité en tant que langage de base.

Puis, dans une deuxième partie de ce chapitre, une autre approche est présentée de façon schématique, sous la forme d'une description très rapide d'un langage, appelé ici L0, sur lequel j'ai travaillé à Grenoble, en particulier pendant le temps où j'avais pris en charge l'encadrement du travail de thèse de D. Bert.

A la fin de cette deuxième partie, ce chapitre se termine sur la suggestion de poursuivre la direction indiquée par L0, en tirant parti également des idées émises par S. Schuman sur la structure d'une implémentation construite autour de ce qu'il appelle un "interpréteur extensible".

3.1. BASEL

Le langage BASEL (BASE Language) a été défini, en 1967, 1968 et 1969 aux Etats-Unis, dans le cadre d'un travail effectué à Massachusetts Computer Associates pour construire un langage extensible appelé ELF (Extensible Language Facility) [Ch68, Jo69b]. C'est à cette époque que j'ai établi la structure générale de BASEL, conçu la plupart de ses traits caractéristiques et construit la totalité de sa définition formelle. Mais il est évident que bien des aspects d'un langage de cette taille sont aussi le fruit de discussions avec d'autres personnes, parmi lesquelles il faut mentionner T.E. Cheatham, A. Fischer, M.M. Hammer et S.A. Schuman. Il ne faut pas non plus ignorer l'influence qu'a eu Algol 68 [vW66, vW67a, vW67b, vW68a, vW68b, Jo69a], car c'est à la même époque que se définissait l'essentiel de ce qui allait être une première version officielle de ce langage. D'autre part, en 1972, au cours du travail de thèse de Madame V. BAJAR, dont la responsabilité m'avait été confiée, j'ai parfois eu l'occasion de "raffiner" certains aspects de BASEL. Les travaux sur ELF ont été parmi les premiers aux Etats-Unis, sur le problème des langages extensibles. ELF devrait être construit avec les deux composants fondamentaux d'un langage extensible : un langage de base et des mécanismes d'extensions. En fait, les travaux sur BASEL, le langage de base, ont tout de suite dominé l'ensemble des activités sur ELF. En effet, dès le début, il avait été décidé que les mécanismes d'extension syntaxique seraient étudiés en dernier, et que l'essentiel de l'effort devait d'abord être porté sur le langage de base et les mécanismes d'extension sémantique. C'est ce qui explique pour une grande part la structure et le contenu de BASEL. En effet, la syntaxe de BASEL est rudimentaire mais BASEL est plus qu'un simple langage de base comme le voudrait en principe l'architecture d'un langage extensible, car il contient lui-même des mécanismes d'extension sémantique élémentaires : un mécanisme de définition de modes, pour définir de nouvelles sortes d'objets, et un mécanisme de définition de procédures génériques, pour définir de nouvelles sortes d'opérations.

Après une première version de BASEL [Ch68, Fi68, Jo69b] qui contenait encore un certain nombre de maladresses, une version finale a été construite dont des descriptions partielles ont été données dans [Jo70a], dans deux rapports techniques de Brown University [Jo70b, By70a] et dans la thèse de Madame V. BAJAR [Ba73]. Une description complète, en version française, en est donnée dans le présent ouvrage.

Avant d'entrer dans cette description, il faut noter que tout le travail de conception de BASEL a été conduit de façon méthodique. En effet, une définition formelle de BASEL a été faite, à l'aide d'un formalisme graphique appelé AMBIT/G, dont le mécanisme est décrit plus loin. Ce qui est important à noter ici à ce sujet, c'est que cette définition formelle a été construite en parallèle avec la conception du langage, qu'il y a eu sans cesse un "feed-back" de la formalisation sur la conception et que cette définition représente de façon réaliste la structure et les algorithmes d'une implémentation. En imposant ainsi un examen systématique et complet des conséquences de chaque détail sur l'ensemble du langage, on a pu isoler et corriger très tôt les incohérences par lesquelles on peut difficilement éviter de passer quand on conçoit un langage de programmation. (On peut remarquer à ce propos que les travaux sur la définition formelle de PL/1 n'ont été entrepris qu'après le fait accompli de la définition "informelle" et quasi-définitive de ce qu'allait être ce langage. Il n'y a donc pas eu ce "feed-back" de la définition formelle vers le langage lui-même. La taille et la complexité de cette définition [Al68a, Al68b, Fl68, Wa68] montrent alors qu'avec le formalisme particulièrement élégant de Vienne [Lu68, Lu69, Ne71, Wn73] on n'est pas parvenu à définir "proprement" PL/1 : cela ne peut être dû qu'à la nature du langage lui-même).

Deux implémentations de BASEL ont été réalisées. La première, aux Etats-Unis, sur matériel Honeywell 516, par moi-même avec la collaboration de P. Moskovites et D. Baeristo, à Mass. Computer Associates dans le cadre d'un travail pour la NASA. Cette implémentation, réalisée en 8 mois, a été une transposition, avec quelques "aménagements" mineurs, de la définition formelle et avait pour but essentiel de montrer qu'un langage de ce niveau était implémentable sans trop de difficultés. La deuxième implémentation, réalisée à Grenoble par Madame V. Bajar, sur matériel IBM 360 modèle 67, avait pour but une étude approfondie des problèmes posés par les langages de la même famille que BASEL, comme Algol 68, surtout dans le domaine des "modes" et de leur manipulation par un compilateur. En effet, BASEL offre un cadre agréable pour l'étude de ces problèmes, alors qu'il n'est pas aisé de bien les isoler en Algol 68. Il avait été envisagé de rendre l'implémentation de Grenoble plus largement disponible à certains groupes d'étudiants, ce qui leur aurait permis d'utiliser un langage avec un mécanisme de modes très évolué, avant la mise en service du compilateur Algol 68. Le départ de Madame Bajar a malheureusement rendu cela impossible à réaliser.

Enfin, si ELF dans son ensemble n'a jamais réellement vu le jour, BASEL a eu un rôle non négligeable dans l'évolution des idées sur les langages de programmation. En particulier, il a été utilisé à l'Université d'Harvard, par le Professeur T.E. Cheatham comme support de son cours sur les principes des langages de programmation. Mais il a surtout servi de point de départ dans la conception de plusieurs autres langages de programmation. En particulier, divers aspects de la structure de BASEL et de son mécanisme de modes ont influencé la conception du langage extensible ECL [We70, Br71, Pr71, Hl71, We71a, We71b] développé à Harvard par B. Wegbreit, dans l'équipe de T.E. Cheatham. D'autre part, à Brown University, D. Berry a défini le langage OREGANO [By70b], avec la collaboration du Professeur P. Wegner qui avait lui-même pris part à de nombreuses discussions sur BASEL avec moi-même et T.E. Cheatham. Ce langage a été fortement influencé, dès le départ, par BASEL. (La thèse de D. Berry [By74], publiée à UCLA, contient une description complète de la version finale d'OREGANO). Enfin, les relations entre BASEL et Algol 68 sont évidentes. L'influence a bien sûr été d'abord d'Algol 68 sur BASEL, mais on retrouve maintenant, dans le langage Algol 68 révisé par le groupe WG 2.1 de l'IFIP, plusieurs aspects qui, quatre ans auparavant étaient propres à BASEL. Cependant, dans un tel cas, il est bien difficile de situer l'origine d'une idée, car Algol 68 a subi beaucoup d'influences avant que ne soit arrêtée sa forme définitive [vW74].

3.1.1. INTRODUCTION A LA DESCRIPTION DE BASEL

La description de BASEL qui est donnée ici est organisée selon un format devenu classique depuis la définition d'Agol 60 [Na60]. Pour chaque construction du langage, il y a un paragraphe sur sa syntaxe, avec éventuellement quelques exemples, et un paragraphe sur sa sémantique, parfois suivi de commentaires.

3.1.1.1. Notations pour décrire la syntaxe

La syntaxe est décrite par un ensemble de règles de grammaire. Chaque règle a la forme :

non-terminal : phrase

où une phrase est une suite de terminaux et de non-terminaux séparés ou regroupés par des symboles métalinguistiques.

- les non-terminaux sont des suites de lettres éventuellement découpées en plusieurs mots par des tirets.
- les terminaux sont des caractères ou des suites de caractères soulignés pour éviter des confusions possibles avec les symboles métalinguistiques.
- les symboles métalinguistiques sont les suivants :

  - le blanc, qui indique la concaténation ;
  - |, qui sert à séparer entre elles les diverses alternatives décrites par une phrase ;
  - [ ], qui indiquent que la phrase incluse est optionnelle ;
  - < >, qui indiquent que la phrase incluse est à considérer dans son ensemble ;
  - ..., qui indiquent qu'il faut prendre une concaténation de 0, 1 ou plusieurs exemplaires de l'élément qui précéde, cet élément pouvant être un non-terminal, un terminal ou une phrase incluse dans < >.

Ainsi, une règle pourra s'écrire :

bloc : [declaration <, declaration> ... dans] partie-instruction

c'est-à-dire qu'un bloc peut ne consister que d'une partie-instruction, mais qu'il peut aussi commencer par une liste de déclarations séparées par des virgules, auquel cas le symbole dans précède immédiatement la partie instruction.

3.1.1.2. Terminologie pour décrire la sémantique

La sémantique est décrite en français, de façon "informelle". Le mécanisme de la définition formelle de BASEL en AMBIT/G étant volontairement introduit plus loin, il a été jugé préférable de donner ici la priorité à une description faisant raisonnablement appel à la culture et à l'intuition du lecteur. Il est cependant nécessaire d'introduire d'abord un certain nombre de termes qui sont utilisés dans cette description.

Deux aspects de la sémantique sont décrits ainsi : son aspect statique et son aspect dynamique. Dans les paragraphes sur la sémantique on trouvera donc, selon le cas, l'un ou l'autre de ces deux aspects, ou les deux.

La sémantique statique a pour but d'établir des règles, relatives à l'écriture des programmes, qui ne sont pas exprimables dans le formalisme syntaxique choisi, comme les règles de compatibilité entre modes ou l'obligation de déclarer un identificateur pour pouvoir l'utiliser.

La sémantique dynamique a pour objet de définir l'enchaînement des actions qui composent ce qu'on appelle habituellement l'exécution ou l'évaluation d'un programme (en Algol 68 : l'élaboration). Ici, cette définition est faite de façon récursive, en donnant pour chaque construction syntaxique de BASEL une définition de sa sémantique composée des actions qui définissent la sémantique de ses propres constituants syntaxiques. Cette composition des actions peut se dérouler dans le temps selon deux organisations qualifiées, comme en Algol 68, de sérielle et de collatéralle.

Une action sert à établir, ou à invalider, des relations entre des objets. Il convient donc, avant tout, de définir ici quels sont les objets impliqués dans la sémantique de BASEL, et quelles sont les relations qui peuvent exister entre eux.

Environnement, activation, origine.

Un environnement est un ensemble totalement ordonné d'activations : on dit qu'une activation est antérieure à une autre. L'activation antérieure à toutes les autres est l'activation initiale, et celle qui n'est antérieure à aucune autre est l'activation actuelle. Pour chaque activation, sauf pour l'activation initiale, il y a donc une activation qui lui est immédiatement antérieure.

Une activation est un ensemble non ordonné d'origines. Chaque origine est identifiée par un identificateur et à chaque origine est associée une valeur. C'est l'opération d'association qui sert à associer une valeur à une origine.

Valeur.

Il y a plusieurs sortes de valeurs. On distingue d'abord les valeurs simples et les valeurs composées. Les valeurs simples peuvent être des valeurs primitives, des variables ou des procédures. Les valeurs composées peuvent être des tuples, des structures ou des séquences. D'autre part, chaque valeur a un mode, qui sert à caractériser la sorte de valeur dont elle fait partie.

Valeurs primitives.

Les valeurs primitives peuvent être de l'un des quatre modes primitifs. En effet, BASEL permet d'utiliser des entiers, des réels, des booléens et des caractères. Les valeurs primitives préexistent toutes et des notations appropriées correspondent à chacune d'elles. Sur ces valeurs, on retrouve en BASEL les opérations et les relations classiques qui existent dans bien d'autres langages de programmation.

Variables.

Une variable est une valeur qui a pour rôle de retenir une autre valeur. Cette autre valeur doit être du mode spécifié dans le mode de la variable. C'est l'opération d'allocation qui permet de créer une variable, et l'opération d'affectation qui fait retenir une valeur par une variable. D'autre part, l'opération d'identité permet de dire si les évaluations respectives de deux expressions ont pour résultat la même variable.

Procédures.

Une procédure est une valeur qui sert à regrouper un certain nombre d'actions. Une procédure est créée par l'évaluation d'un texte de procédure, et l'exécution des actions qu'elle regroupe est provoquée par l'appel de cette procédure, qui peut être accompagné d'un passage de paramètres. Le mode d'une procédure spécifie, dans l'ordre, le mode que doit avoir chacun des paramètres éventuels et, si une valeur est attendue comme résultat de l'appel, il indique aussi le mode de ce résultat.

Tuples.

Un tuple est un ensemble ordonné de valeurs, appelées ses éléments.
Le mode d'un tuple spécifie, dans l'ordre, le mode de ses éléments.
Un tuple est créé par l'évaluation d'une expression composée.

Structures.

Une structure est un ensemble ordonné de valeurs, appelées ses champs.
A chaque champ est attaché un sélecteur qui est un identificateur dont le rôle est de permettre d'isoler ce champ grâce à une opération de sélection. Le mode d'une structure spécifie, dans l'ordre, le mode de ses champs et, pour chaque champ, le libellé de son sélecteur. Une structure est toujours construite à partir d'un tuple.

Séquences.

Une séquence est un ensemble ordonné de variables qui sont toutes du même mode. Le mode d'une séquence spécifie le mode des valeurs qui peuvent être retenues par les variables qui la composent. C'est l'opération d'itération qui est utilisée pour créer une séquence, mais il est également possible d'obtenir une séquence à partir d'un tuple. Les variables qui composent une séquence sont en nombre quelconque, et sont numérotées de 1 à la longueur de la séquence. On peut modifier la longueur d'une séquence grâce à l'opération de changement. Chaque variable individuelle dans une séquence peut être utilisée indépendamment des autres, comme une variable normale, en l'obtenant par indexation de cette séquence. L'indexation permet aussi d'isoler des sous-séquences d'une séquence.

Le mode "union".

En BASEL, une valeur, c'est-à-dire un entier, un réel, un booléen, un caractère, une variable, une procédure, un tuple, une structure ou une séquence, peut être rencontrée dans un certain nombre de situations générales : associée à une origine, retenue par une variable, élément d'un tuple, champ d'une structure, retenue par une variable elle-même contenue dans une séquence, résultat d'un appel de procédure et, en général, résultat de l'évaluation d'une expression. Dans chaque cas, la situation particulière où la valeur est rencontrée permet de déterminer le mode de cette valeur.
Cependant, il est possible d'autoriser la présence, dans chacune des situations générales ci-dessus, d'objets qui ne sont plus eux-mêmes des valeurs d'un mode particulier, mais qui sont susceptibles de prendre des valeurs dont le mode peut être différent à divers instants de l'exécution. Ces objets sont assimilés à des valeurs, car on les rencontre dans les mêmes situations générales, et ils ont eux-mêmes un mode, qui spécifie les modes possibles pour les valeurs qu'ils peuvent prendre : un tel mode est un mode union. C'est la déclaration conditionnelle qui permet de contrôler quel est le mode de la valeur prise à un instant donné par un objet de mode union.

3.1.2. DESCRIPTION DE BASEL

A. Le programme

A.1. Syntaxe

programme : expression-composée

A.2. Sémantique

A.2.1. Statique

L'expression composée qui constitue le programme doit avoir un résultat vide.

A.2.2. Dynamique

L'exécution du programme est l'exécution de l'expression composée qui le constitue.

B. Les expressions-composées

B.1. Syntaxe

expression-composée : expression-composée-sérielle
expression-composée-collatéralle
expression-composée-sérielle : (corps-d-expression-composée)
expression-composée-collatéralle : [corps-d-expression-composée]
corps-d-expression-composée : bloc <, bloc> ...

B.2. Sémantique

B.2.1. Statique

Soient b1, b2, ..., bn les blocs qui constituent, dans cet ordre, le corps de l'expression composée. Parmi ces blocs, bi1, bi2, ..., bip, avec i1<i2<...<ip, ont des résultats non vides, de modes respectifs mi1, mi2, ..., mip, les autres blocs ayant tous des résultats vides.

Si p = 0, l'expression composée est à résultat vide.
Si p = 1, et si l'expression composée est sérielle, son résultat est de mode mi1.
Si p = 1, et si l'expression composée est collatéralle, son résultat est de mode tuple (mi1)
Si p > 1, l'expression composée a un résultat de mode tuple (mi1, mi2, ..., mip)

B.2.2. Dynamique

Si l'expression composée est sérielle, son exécution est composée de façon sérielle à partir des exécutions respectives des blocs b1, b2, ..., bn.
Si l'expression-composée est collatéralle, son exécution est composée de façon collatéralle à partir des exécutions respectives des blocs b1, b2, ..., bn.

B.2.3. Commentaires

Les blocs qui constituent le corps de l'expression composée sont non adjacents, c'est-à-dire que si i ≠ j, les déclarations faites dans bi sont inconnues dans bj et vice-versa.

D'autre part, on peut raisonnablement avancer que les cas les plus fréquents en programmation courante sont ceux où p = 0 et où p = n.

Lorsque p = 0, on voit qu'il s'agit de l'exécution de n blocs indépendants, qui ne retournent pas de résultat. Cette indépendance systématique entre les blocs, conséquence de l'organisation générale de BASEL, favorise une saine structuration des programmes et entraîne aussi une utilisation économique de la mémoire. (Les blocs d'un corps d'expression composée peuvent néanmoins communiquer entre eux par des objets déclarés dans des blocs qui les englobent).

Lorsque p = n, avec n > 1, il s'agit de la construction d'un tuple dont l'usage peut être fort divers. D'une façon générale, un tuple sert à rassembler un certain nombre de valeurs destinées à un usage déterminé, comme construire une structure ou fournir des paramètres effectifs à un appel de procédure.

De plus, lorsque p = 1, c'est-à-dire lorsqu'un seul bloc a un résultat non vide, on aura aussi très souvent n = 1, c'est-à-dire que ce bloc sera le seul dans l'expression composée. Il devient alors sans objet de distinguer, dans ce cas, entre exécution sérielle et exécution collatéralle. Comme il faut pouvoir construire des tuples à un seul élément, c'est la notation [b] qui a été retenue pour remplir ce rôle, laissant ainsi à la notation (b) la signification habituelle des parenthèses. Enfin, il est intéressant de rapprocher ces conventions de celles relatives à la transmission des paramètres lors d'un appel de procédure. Pour appeler une procédure avec paramètres (voir le paragraphe K), deux cas peuvent se présenter : cette procédure requiert plusieurs paramètres effectifs, ou elle ne requiert qu'un seul paramètre effectif. Soient e, e1, e2, ..., en des expressions susceptibles de calculer les paramètres effectifs et p l'expression qui a pour résultat la procédure à appeler. Dans le premier cas, il y a lieu de rassembler plusieurs paramètres, en écrivant l'appel sous l'une des formes p(e1, e2, ..., en) ou p [e1, e2, ..., en] selon qu'on veut évaluer les paramètres de façon sérielle ou collatéralle, et c'est la valeur du tuple à plusieurs éléments qui est transmise à la procédure. Dans le second cas, il n'y a évidemmment pas lieu de rassembler quoi que ce soit. On écrit donc p e, ou p(e), ou p((e)), ... et c'est la valeur de e qui est transmise comme paramètre à la procédure.

Ainsi, en BASEL, ce sont les mêmes parenthèses, celles de l'expression composée, qui tiennent le rôle de délimiteur de bloc, de regroupement dans un but de priorité d'opération et de parenthèses pour une liste de paramètres. On verra qu'elles ont aussi le rôle du si ... fsi d'Algol 68.

Cependant, il ne faut pas non plus ignorer l'utilité des cas où $p \neq 0$ avec $p < n$. En effet, d'une façon générale, une expression composée permet d'insérer autant de blocs à résultat vide que l'on veut entre les blocs qui calculent les éléments de son résultat. Par exemple, on peut imaginer qu'après avoir obtenu les paramètres effectifs d'un appel de procédure, mais avant de les transmettre à la procédure, on veuille exécuter un certain nombre de blocs qui préparent le travail de cet appel particulier. Tout cela peut être contenu dans la même expression composée sérielle, celle qui a pour résultat le tuple des paramètres. Un autre exemple qui fait usage de cette caractéristique des expressions composées est le suivant :

X = (Y, Y = X ; vide)

Ceci a pour effet d'échanger les valeurs associées aux origines identifiées respectivement par X et par Y. En effet, l'expression composée utilisée ici a deux blocs :

Y

et Y = X ; vide

Le premier a pour résultat la valeur associée à Y. Le second a un résultat vide, mais il commence par associer à Y la valeur associée à X. L'expression composée a donc pour résultat la valeur associée primitivement à Y, qui est finalement associée à X.

C. Les blocs

C.1. Syntaxe

bloc : [declaration <, declaration> ... dans] partie-instruction

C.2. Sémantique

C.2.1. Statique

Lorsqu'un bloc contient des déclarations, celles-ci ont une portée statique, dans le texte du programme, qui obéit à des règles analogues à celles que l'on trouve en Algol 60, les divers niveaux de l'imbrication des blocs étant ici soit des expressions composées, soit des textes de procédures.

Quand, parmi ses déclarations, un bloc ne contient pas de déclaration conditionnelle, le mode de son résultat est celui du résultat, s'il n'est pas vide, de sa partie instruction. Si sa partie instruction est à résultat vide, le bloc est lui aussi à résultat vide.

Quand, parmi ses déclarations, un bloc contient au moins une déclaration conditionnelle, son résultat est vide.

C.2.2. Dynamique

L'exécution d'un bloc se fait en trois phases successives, appelées prologue, corps et épilogue. Le prologue concerne les déclarations et se déroule lui-même en deux temps. Dans un premier temps sont exécutées les déclarations conditionnelles (voir le paragraphe D), s'il y en a. Si au moins l'une d'entre elles n'est pas vérifiée, ni le corps ni l'épilogue n'ont lieu et l'exécution du bloc est terminée. Si toutes les déclarations conditionnelles sont vérifiées, ou s'il n'y en a pas, le traitement des déclarations entre dans un deuxième temps, où est créée une nouvelle activation, qui devient l'activation actuelle. Cette activation est constituée des origines créées par les déclarations d'origine faites dans ce bloc, c'est-à-dire les déclarations simples et les déclarations additionnelles (voir le paragraphe D).

Le corps de l'exécution du bloc est l'exécution de sa partie instruction. Si le bloc n'a pas de déclaration d'origine, son exécution est alors terminée dès que le corps de l'exécution est lui-même terminé.

L'épilogue de l'exécution du bloc n'a donc lieu que si ce bloc a des déclarations d'origine, et il a pour rôle de supprimer l'activation actuelle créée dans le prologue du bloc et de promouvoir à cette fonction celle qui lui était immédiatement antérieure.

C.2.3. Commentaires

La structure syntaxique des blocs en BASEL impose que tout soit déclaré avant d'être utilisé dans la partie instruction. Cependant, à l'intérieur même de la liste de déclarations, l'ordre d'écriture est indifférent et sans signification. Ainsi, Algol 68-R [Cu71] a rejoint BASEL pour la règle de déclaration avant utilisation, mais BASEL est resté moins strict car il n'y est pas fait usage d'une déclaration du genre "mode m" comme cela est nécessaire pour les modes récursifs en Algol 68-R.

On peut d'ailleurs noter à ce propos que la raison de ces restrictions en Algol 68-R est à chercher uniquement dans le choix d'une méthode de compilation en un seul passage. BASEL se compile aussi en un seul passage mais chaque partie déclaration doit être examinée deux fois.

D'autre part, le fait qu'un bloc soit à résultat vide quand il contient au moins une déclaration conditionnelle est dû au principe même de telles déclarations : un bloc qui en contient au moins une peut ne pas être exécuté du tout. On ne peut donc pas garantir, à la compilation, que son résultat éventuel sera obtenu. Le principe de sécurité dans la conception du langage conduit donc à supposer que ce résultat ne peut pas être utilisable, donc à considérer qu'il n'existe pas.

D. Les déclarations

D.1. Syntaxe

déclaration : déclaration-de-mode |
déclaration-de-générique |
déclaration-d-origine |
déclaration-conditionnelle

déclaration-de-mode : symbole rep mode

déclaration-de-générique : identificateur est générique

déclaration-d-origine : déclaration-simple
déclaration-additionnelle

déclaration-simple : identificateur est mode

déclaration-additionnelle : identificateur estaussi mode

déclaration-conditionnelle : identificateur doitêtre mode

EXEMPLES

complexe rep struct (reel pr, reel pi)

matrice rep seq seq ent

plus est générique

x est reel

z est var complexe

u est union (ent, bool)

plus estaussi proc (complexe, complexe) complexe

plus estaussi proc (matrice, matrice) matrice

u doitêtre ent

D.2. Sémantique

D.2.1. Statique

Les déclarations de modes et les déclarations de génériques ont un rôle uniquement statique.
Les déclarations simples, les déclarations additionnelles et les déclarations conditionnelles ont à la fois un rôle statique et un rôle dynamique.

Les déclarations de modes ont en BASEL un rôle et un fonctionnement très proches de ceux qui sont les leurs en Algol 68. Déclarer un mode sert à définir, pour la portée statique délimitée par le bloc où est faite cette déclaration, un nouveau symbole qui représente le mode écrit dans la déclaration, (dans ces pages, un symbole est mis sous la forme d'un identificateur souligné) c'est-à-dire que dans cette portée statique, le symbole déclaré peut être utilisé comme synonyme du mode qu'il représente. Les déclarations récursives sont autorisées, mais il faut alors que soit respectée la règle sur les "modes bien formés", qui est définie au paragraphe E.2.2.

Une déclaration de générique sert à définir, pour la portée statique délimitée par le bloc où est faite cette déclaration, un identificateur appelé procédure générique. La notion de procédure générique n'a pas été définie dans les paragraphes d'introduction sur les objets de la sémantique dynamique car il ne s'agit pas d'un objet manipulé par des actions, mais d'un identificateur susceptible d'identifier un ensemble d'origines auxquelles ne peuvent être associées que des procédures avec paramètres. C'est par des déclarations additionnelles que sont définies et identifiées individuellement chacune des origines de l'ensemble identifié par la procédure générique.

Une déclaration simple sert à définir, pour la portée statique délimitée par le bloc où est faite cette déclaration, un identificateur et l'origine qu'il identifie. Les valeurs qui pourront être associées à cette origine devront être du mode spécifié dans la déclaration.

Le mode spécifié dans une déclaration additionnelle doit être un mode de procédure avec paramètres.

Une déclaration additionnelle sert à définir, pour la portée statique délimitée par le bloc où est faite cette déclaration, d'une part l'une des origines faisant partie de l'ensemble d'origines identifié par une procédure générique, d'autre part une façon d'identifier cette origine. L'identificateur utilisé dans une déclaration additionnelle doit être une procédure générique et c'est de cette procédure générique que la déclaration additionnelle définit l'une des origines, à laquelle peut être associée une procédure. Les déclarations additionnelles relatives à une procédure générique donnée peuvent donc être faites dans l'un quelconque des blocs et textes de procédures couverts par la portée statique de cette procédure générique, y compris dans le bloc où elle est elle-même déclarée. Là encore, il faut noter que l'ordre d'écriture des déclarations dans la liste de déclarations d'un bloc est sans importance.

Par exemple, on peut avoir :

```
(p est générique, p estaussi ..., p estaussi ... dans ...
     (p estaussi ... dans ...
          (p estaussi ..., p estaussi ... dans ...) ...) ...)
```

Chaque déclaration additionnelle définit ainsi une origine qui est identifiée en deux étapes : d'abord, de façon "générique" par la procédure générique dont elle fait partie, puis, de façon individuelle, par la liste des modes des paramètres acceptés par les procédures qui pourront lui être associées. Les ambiguités d'identification que ce mécanisme peut entraîner sont étudiées plus loin, à propos de l'association d'une procédure à l'une des origines d'une procédure générique (voir le paragraphe G.2.1.) et à propos de l'appel de l'une de ces procédures (voir le paragraphe K.2.1.1.).

Une déclaration conditionnelle sert à contrôler le mode de la valeur prise par un objet de mode union associé à une origine et à permettre une utilisation de cette origine comme si lui étaient associables seulement des valeurs du mode contrôlé, et non plus des objets de son mode union original, pour la portée statique délimitée par le bloc où est faite cette déclaration. L'identificateur utilisé dans une déclaration conditionnelle doit donc identifier une origine à laquelle peuvent être associés des objets de mode union.

Exemple :

```
(a est union (ent, bool) dans ...
        (a doitetre ent, ... dans ...) ...
        (a doitetre bool dans ...) ...)
```

La partie du programme dans laquelle on peut faire des déclarations conditionnelles portant sur une origine donnée est donc contenue dans la portée statique de la déclaration simple de cette origine, mais elle ne couvre pas toute cette portée. D'une part, en est exclus le bloc où cette origine est elle-même déclarée, car il serait sans fondement de la déclarer et de contrôler le mode d'une valeur qui lui serait associée, dans la même liste de déclaration, avant même de lui avoir associé une valeur d'un mode particulier. D'autre part, en sont exclus tous les textes de procédures inclus dans cette portée, car une procédure ne peut pas faire usage d'une origine déclarée à son extérieur, mais seulement de la valeur associée à cette origine. Tous les détails sur ce phénomène sont donnés au paragraphe M.2.2.2. sur l'évaluation des textes de procédure (à ne pas confondre avec l'appel d'une procédure). De plus, le mode spécifié dans une déclaration conditionnelle ne peut pas être quelconque. En effet, ce mode doit être compatible avec le mode union des objets associables à l'origine utilisée, au sens de la relation de compatibilité entre les modes définies au paragraphe E.2.4.. Dans la partie instruction du bloc où est faite une déclaration conditionnelle relative à une origine, toute utilisation de cette origine ou de la valeur qui lui est associée est considérée comme étant du mode spécifié dans la déclaration conditionnelle.

Enfin, il ne faut pas qu'il y ait de double déclaration dans une même liste de déclarations :

- un symbole ne peut être déclaré qu'une fois, par déclaration de mode, dans une même liste de déclarations.
- un identificateur ne peut être déclaré qu'une fois, par déclaration simple, conditionnelle ou de générique, dans une même liste de déclarations.

Par contre, une procédure générique peut être utilisée un nombre quelconque de fois par des déclarations additionnelles dans la même liste de déclarations, à condition que le même identificateur n'y soit pas aussi déclaré par déclaration simple ou conditionnelle.

D.2.2. Dynamique

Une déclaration simple est exécutée dans le deuxième temps du prologue de l'exécution du bloc dont elle fait partie. Elle crée une origine, élément de l'activation du bloc, à laquelle pourront être associées des valeurs du mode spécifié. Pendant le corps de l'exécution du bloc, c'est cette origine qui est identifiée par l'identificateur utilisé dans la déclaration. Avant que, dans le corps de l'exécution, ne soit associée à cette origine une valeur particulière, c'est la valeur nulle du mode spécifié (voir le paragraphe N) qui lui est associée .

Une déclaration additionnelle est exécutée dans le deuxième temps du prologue de l'exécution du bloc dont elle fait partie. Elle crée une origine, élément de l'activation du bloc, à laquelle pourront être associées des procédures du mode spécifié. Pendant le corps de l'exécution du bloc, c'est cette origine qui est identifiée par la procédure générique accompagnée de la liste des modes des paramètres spécifiés dans la déclaration. Avant que, dans le corps de l'exécution, ne soit associée à cette origine une procédure particulière, c'est la valeur nulle du mode spécifié qui lui est associée.

Une déclaration conditionnelle est exécutée dans le premier temps du prologue de l'exécution du bloc dont elle fait partie. Si le mode de la valeur prise par l'objet de mode union associé à l'origine identifiée par l'identificateur utilisé dans la déclaration est égal (voir le paragraphe E.2.3.) au mode spécifié par la déclaration, la déclaration conditionnelle est vérifiée. L'identificateur identifie alors l'origine de façon à ce qu'elle soit utilisée dans le corps de l'exécution comme une origine à laquelle ne peuvent être associées que des valeurs du mode spécifié dans la déclaration.

D.2.3. Commentaires

La déclaration simple ressemble beaucoup aux déclarations tout à fait habituelles comme en avait déjà Algol 60. Cependant, ce n'est pas vraiment une variable, au sens classique, que l'on y déclare. En effet, à l'origine créée par une déclaration simple ne peuvent pas être associées des valeurs avec la même liberté que des valeurs peuvent être affectées à des variables.

Ceci est dû au fait que les textes de procédures écrits dans la portée d'une déclaration simple ne peuvent pas faire usage de l'origine créée par cette déclaration, mais seulement de la valeur qui lui est associée au moment où ce texte est évalué (voir le paragraphe M.2.2.2.).

La déclaration de générique remplit l'une des deux fonctions remplies en Algol 68 par la déclaration de priorité. La déclaration de priorité a en effet un double rôle : elle définit un symbole, et elle attache à ce symbole une information syntaxique de priorité. Les problèmes d'extension syntaxique n'étant pas abordés en BASEL - même à un niveau aussi élémentaire qu'une priorité d'opérateur - seul l'aspect sémantique a été retenu ici : la déclaration de générique définit un identificateur qui peut prendre l'une ou l'autre des significations qui lui sont attachées par des déclarations additionnelles, selon le nombre et le mode des paramètres des appels où il joue le rôle de procédure.

On voit alors que la déclaration additionnelle est analogue à la déclaration d'opération en Algol 68, mais elle est plus générale car rien ne limite ici à deux le nombre des paramètres de chaque procédure, et pour une même procédure générique, deux déclarations additionnelles peuvent spécifier des modes de procédures n'ayant pas le même nombre de paramètres.

Enfin, la déclaration conditionnelle apporte une solution originale au problème du test dynamique des modes. En Algol 68, la solution retenue est le cas de conformité, qui a hérité de la relation conformité son principe de base : lors d'un test dynamique de mode, il y a d'abord une comparaison du mode de la valeur prise par l'objet de mode union avec le mode demandé dans le test, puis, si l'on envisage d'utiliser la valeur prise par l'objet de mode union au cas où la comparaison aurait été satisfaite, le test est suivi d'une recopie de cette valeur, et c'est la copie que l'on utilise. Le coût en temps et en espace d'une telle opération peut devenir gênant si la taille de la valeur est grande, s'il s'agit par exemple d'un tableau ou d'une structure avec de nombreux champs. D'autre part, on peut estimer que ce mécanisme n'est pas "naturel", car il impose, en général, de déclarer deux identificateurs pour accèder à ce que l'on veut considérer, en fait, comme étant la même valeur. Ceci a également pour conséquence que les changements apportés à la valeur copiée ne sont évidemment pas reflétés sur son original. En BASEL, la déclaration conditionnelle évite tous ces écueils et permet de dire quelque-chose comme : "Si l'objet x de mode union est main-

tenant de mode m, alors utilisons le exactement comme un objet de mode m". En effet, la déclaration conditionnelle évite toute manipulation préalable de la valeur prise par l'objet de mode union associé à une origine et permet une utilisation directe de cette valeur. Dans la portée statique d'une déclaration conditionnelle, l'identificateur qui identifie cette origine l'identifie de façon à ce que la valeur associée ne soit plus l'objet de mode union dans son ensemble mais seulement la valeur qui est du mode faisant l'objet du test dynamique. Il ne reste donc plus, de façon dynamique, qu'à effectuer la comparaison de modes. D'autre part, la structuration en expressions composées que propose BASEL se prête bien à l'utilisation de cette déclaration conditionnelle et permet de retrouver une organisation analogue à celle du cas de conformité :

```
(u est union (ent, bool, car) dans ...
        ... (u doitêtre ent dans ...,
             u doitêtre bool dans ...,
             u doitêtre car dans ...) ...)
```

On remarque alors la clarté de la structure du programme obtenu, qui est due en particulier à la nécessité d'utiliser le même identificateur u que celui qui identifie l'origine à laquelle est associé l'objet de mode union.

E. Les modes

E.1. Syntaxe

```
mode : primitif|
       variable|
       procédure|
       tuple|
       structure|
       séquence|
       union|
       symbole
primitif : ent|reel|bool|car
variable : var mode
procédure : proc [( mode <, mode> ...)] <mode|rien>
tuple : tuple (mode <, mode> ...)
```

structure : struct (mode identificateur <, mode identificateur> ...)
séquence : seq mode
union : union (mode <, mode> ...)

EXEMPLES :

ent
var bool
proc (union (ent, bool)) car
tuple (complexe, bool)
struct (chaîne nom, bool sexe, ent age)

E.2. Sémantique

E.2.1. Statique

Comme en Algol 68, la notion de mode est utilisée en BASEL pour contrôler statiquement que chaque objet est employé correctement, conformément à ses propriétés, et que chaque opération reçoit bien comme opérandes des valeurs de nature correspondant à ce qu'elle attend. Le mode d'une valeur exprime donc, de façon conventionnelle, la nature et les propriétés de cette valeur.

Soient donc m, m1, m2, ..., mn des modes et s1, s2, ..., sn des identificateurs.

ent est le mode des entiers.
reel est le mode des réels.
bool est le mode des booléens.
car est le mode des caractères.
var m est le mode des variables susceptibles de retenir une valeur de mode m.
proc (m1, m2, ..., mn) m est le mode des procédures à n paramètres, de modes respectifs m1, m2, ..., mn, et à résultat de mode m.
proc m est le mode des procédures sans paramètre et à résultat de mode m.
proc (m1, m2, ..., mn) rien est le mode des procédures à n paramètres, de modes respectifs m1, m2, ..., mn, et à résultat vide.
proc rien est le mode des procédures sans paramètre et à résultat vide.

tuple (m1, m2, ..., mn) est le mode des tuples à n éléments de modes respectifs m1, m2, ..., mn.

struct (m1 s1, m2 s2, ..., mn sn) est le mode des structures à n champs, de modes respectifs m1, m2, ..., mn, et auxquels sont attachés respectivement les sélecteurs s1, s2, ..., sn.

seq m est le mode des séquences composées de variables susceptibles de retenir des valeurs de mode m.

union (m1, m2, ..., mn) est le mode des objets de mode union susceptibles de prendre des valeurs de mode soit m1, soit m2, ..., soit mn.

Le mécanisme automatique qui contrôle statiquement que chaque objet sera utilisé dynamiquement conformément à son mode est construit à partir de deux relations élémentaires qui peuvent exister entre deux modes.

Soient deux modes m et n. Ces relations sont :

- l'égalité, notée m =n.
- la compatibilité, notée m ⇒ n si m est compatible avec n.

Mais il faut avant tout, pour pouvoir établir ces relations elles-mêmes, que les modes qu'elles mettent en jeu soient bien formés.

E.2.2. Modes bien formés

La syntaxe de BASEL spécifie les endroits où il est possible d'écrire un mode. Ce sont essentiellement :

- dans une liste de déclarations, pour les déclarations de modes, simples, conditionnelles et additionnelles.
- dans les modifications explicites (voir le paragraphe L).
- dans les notations de valeurs nulles (voir le paragraphe N).
- et, évidemment, à l'intérieur d'un mode commençant par var, proc, tuple, struct, seq ou union.

D'autre part, en chacun de ces endroits, un mode peut être simplement mis sous la forme d'un symbole : ce symbole doit alors avoir fait l'objet d'une déclaration de mode, et il représente le mode spécifié dans sa déclaration. On peut donc, à l'intérieur d'une même liste de déclarations, avoir une déclaration de mode qui définit un symbole et une déclaration de mode qui fait usage de ce symbole dans le mode qu'elle spécifie. Ce peut même être la même déclaration : il est donc possible, comme en Algol 68, grâce à la déclaration de mode, de construire des modes de façon récursive.

Exemples :

Une arborescence avec une valeur entière attachée à chaque noeud peut être représentée par des objets ayant le mode :

arbre rep struct (ent noeud, seq arbre branches)

Les structures de liste de LISP peuvent être représentées par des objets de mode :

liste rep var tuple (elemliste, elemliste)
elemliste rep union (atome, liste)
atome rep car

Mais la structure des modes récursifs ne peut pas être quelconque. En effet, des déclarations comme :

m rep m
ou m1 rep m2,
m2 rep m1

sont vides de sens. D'autre part, certains modes pourraient correspondre à des valeurs qu'il est impossible de construire dans un espace fini, comme :

m rep tuple (ent, m)

En effet, si on admet qu'un tuple est construit sous la forme d'une juxtaposition de ses éléments, un objet de ce mode m occupe un espace infini : un entier, puis un objet de mode m, c'est-à-dire un entier, puis un objet de mode m, ...

Enfin, certains modes seraient ceux d'objets qui, même si on peut les construire, auraient une utilité douteuse, comme :

m1 rep var m1
ou m2 rep seq m2
ou m3 rep proc (m3),ent

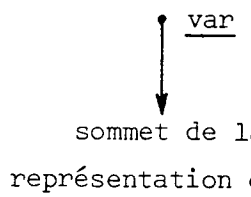
Un mode doit donc être bien formé, selon les définitions qui suivent.

E.2.2.1. Définitions préliminaires

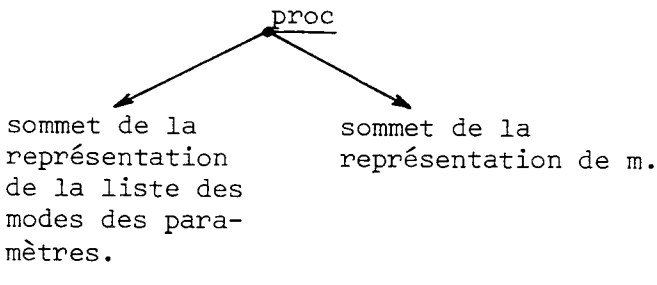
Un mode peut être représenté sous la forme d'un graphe orienté, image concise de la façon dont il est construit, et dont la configuration obéit aux règles suivantes :

- les modes primitifs sont représentés par des noeuds, appelés sommets de leur représentation, qui portent respectivement les libellés ent, reel, bool et car :

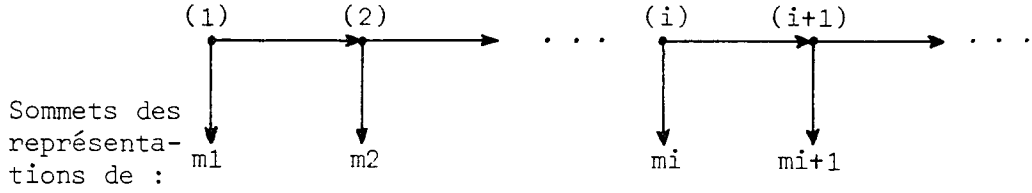
- le mode var m est représenté par un noeud, sommet de sa représentation, qui porte le libellé var, d'où part un arc vers le sommet de la représentation du mode m :



- le mode proc (m1, m2, ..., mn) m est représenté par un noeud, sommet de sa représentation, qui porte le libellé proc et d'où partent deux arcs, l'un allant vers le sommet de la représentation de la liste des modes des paramètres, et l'autre allant vers le sommet de la représentation de m :

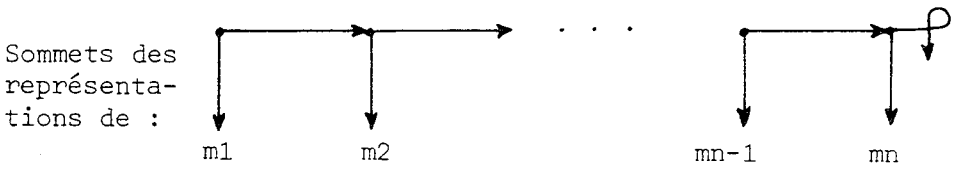


- une liste de n modes m1, m2, ..., mn est représentée par n noeuds, dont le premier est le sommet de la représentation de la liste, et de chaque noeud partent deux arcs : du $i^{ème}$ noeud ($1 \leq i \leq n-1$) partent un arc qui va vers le sommet de la représentation de mi et un arc qui va vers le $(i+1)^{ème}$ noeud :

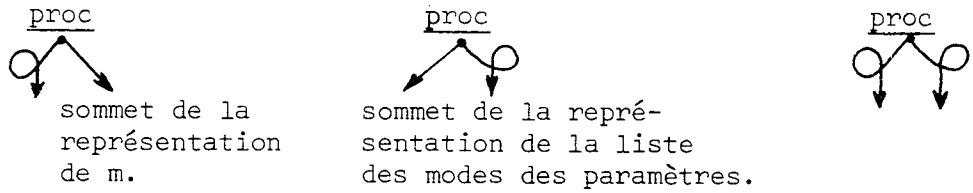


Du $n^{ème}$ noeud partent un arc qui va vers le sommet de la représentation de mn et un arc qui va "vers le vide" (c'est le NIL de LISP). Un arc partant d'un noeud pour aller vers le vide est représenté graphiquement par :

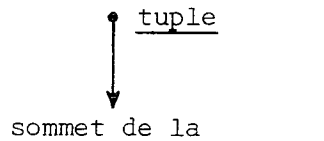
Donc, une liste de modes a la structure :



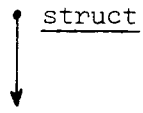
- S'il n'y a pas de paramètres, ou si le résultat est vide (symbole rien dans le mode) les arcs correspondants qui partent du noeud libellé proc vont vers le vide :



- le mode tuple (m1, m2, ..., mn) est représenté par un noeud, sommet de sa représentation, qui porte le libellé tuple et d'où part un arc qui va vers le sommet de la représentation de la liste des modes des éléments :

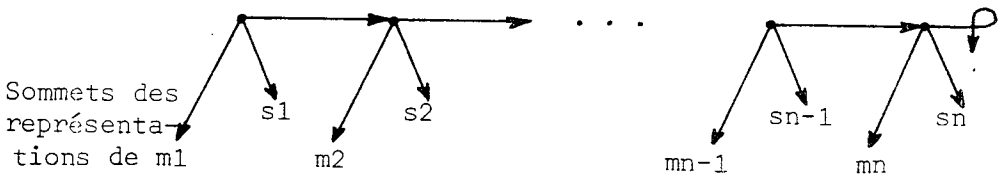


- le mode struct (m1 s1, m2 s2, ..., mn sn) est représenté par un noeud portant le libellé struct, sommet de la représentation, d'où part un arc qui va vers le sommet de la représentation de la liste des champs :

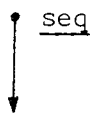


sommet de la représentation de la liste des champs.

- une liste de champs est représentée selon des conventions analogues : celles appliquées à une liste de modes, mais du $i^{ème}$ noeud partent trois arcs, un vers le $(i+1)^{ème}$ noeud, un vers le sommet de la représentation de mi, un vers la représentation (non détaillée ici) du sélecteur si :



- le mode seq m est représenté par un noeud portant le libellé seq, sommet de la représentation, d'où part un arc qui va vers le sommet de la représentation de m :



sommet de la représentation de m.

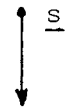
- le mode union (m1, m2, ..., mn) est représenté par un noeud portant le libellé union, sommet de la représentation, d'où part un arc qui va vers le sommet de la représentation de la liste des modes possibles

sommet de la représentation de la liste des modes.

- lorsqu'un symbole est défini par une déclaration de mode :

  s rep m

  ceci est représenté par un noeud portant le libellé du symbole, d'où part un arc qui va vers le sommet de la représentation de m. Il n'y a qu'un seul noeud portant le libellé d'un symbole donné :



sommet de la
représentation de m.

- lorsqu'un symbole est utilisé, la représentation qui correspond à cette utilisation peut être mise sous deux formes différentes :
  - . son utilisation peut être représentée par le noeud unique qui porte le libellé de ce symbole. C'est la forme temporaire de la représentation de l'utilisation d'un symbole.
  - . son utilisation peut être représentée par le sommet de la représentation du mode, extrémité de l'arc dont l'origine est le noeud unique qui porte le libellé de ce symbole. C'est la forme définitive de la représentation de l'utilisation d'un symbole.

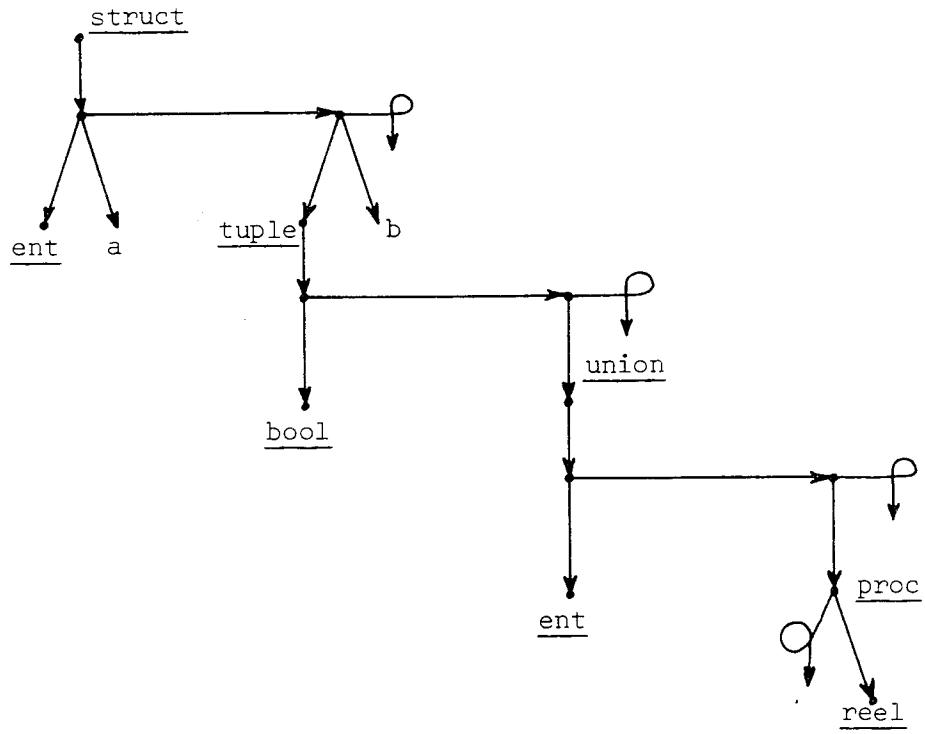
Lorsque, dans la représentation d'un mode, les représentations des symboles utilisés sont toutes sous forme temporaire, la représentation de ce mode est dite temporaire.

Lorsque, dans la représentation d'un mode, les représentations des symboles utilisés sont toutes sous forme définitive, la représentation de ce mode est dite définitive.

E.2.2.2. Exemples de représentations

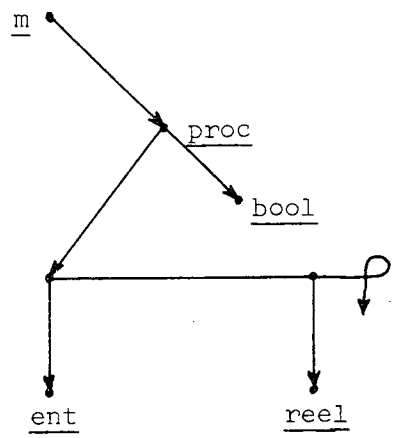
struct (ent a, tuple (bool, union (ent, proc reel)) b)

a la représentation -temporaire et définitive- suivante :



m rep proc (ent, reel) bool

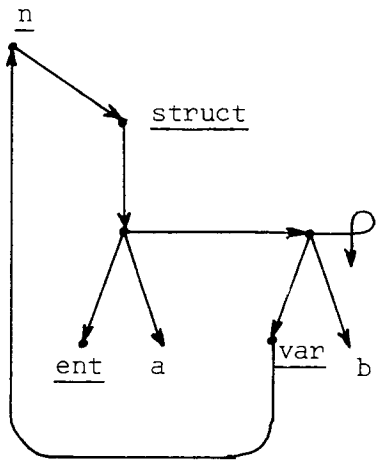
a la représentation -temporaire et définitive- suivante :



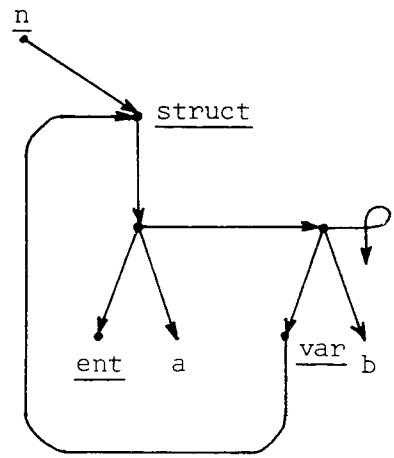
n rep struct (ent a, var n b)

a les représentations suivantes :

temporaire :



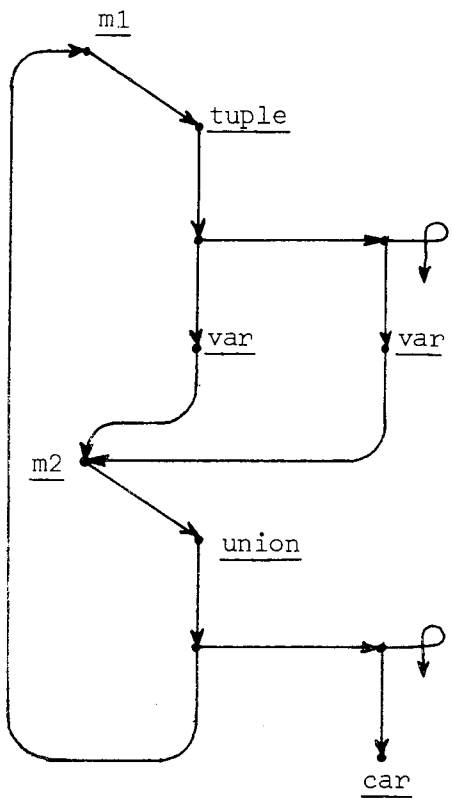
définitive :



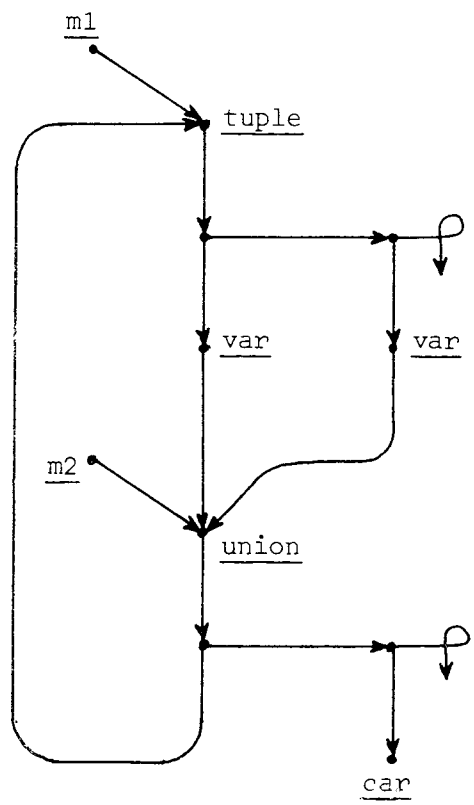
m1 rep tuple (var m2, var m2),
m2 rep union (m1, car)

ont les représentations suivantes :

temporaires :



définitives :



L'après ces définitions, il est clair qu'un mode non récursif est représenté par un arbre, et que tout mode construit de façon récursive et représenté par un graphe contenant au moins un circuit. Dans la représentation temporaire, le noeud représentant le symbole défini par la déclaration qui crée un mode récursif est sur ce circuit.

C'est un examen des représentations temporaires, et en particulier de leurs circuits éventuels, qui permet de décider si les modes sont bien formés.

Tout d'abord, le problème de savoir si un mode est bien formé est trivial si ce mode n'est pas récursif. Donc, si la représentation temporaire d'un mode ne contient pas de circuit, ce mode est bien formé.

Si le mode est récursif, il faut effectuer un contrôle de la composition des circuits contenus dans sa représentation temporaire : il faut que ces circuits soient eux-mêmes bien formés. Soit donc un circuit contenu dans la représentation temporaire d'un mode et s un symbole dont le libellé est porté par l'un des noeuds de ce circuit : ce noeud est appelé ici le noeud s. Ainsi, partant du noeud s, un parcours du circuit revenant au noeud s passe par un certain nombre de noeuds dont certains portent des libellés. Dans l'ordre du parcours, ces libellés sont :

$$s, \ell 1, \ell 2, \ldots, \ell n, s$$

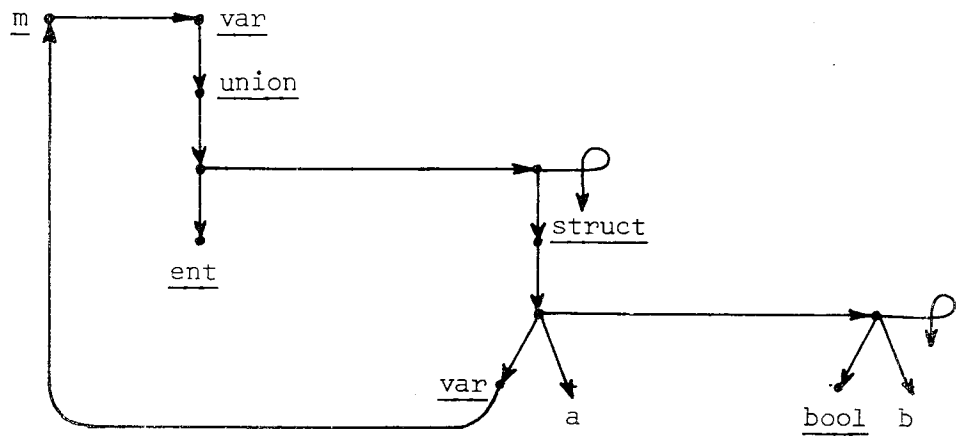
soit ℓ la suite :

$$\ell = \ell 1, \ell 2, \ldots, \ell n$$

Ainsi, si l'on a déclaré :

m rep var union (ent, struct (var m a, bool b))

la représentation temporaire de ce mode est :



En partant du noeud m, un parcours du seul circuit de cette représentation passe par des noeuds dont la suite des libellés est :

$$\ell = \underline{var}, \underline{union}, \underline{struct}, \underline{var}$$

Dans la suite $\ell = \underline{\ell 1}, \underline{\ell 2}, \ldots, \underline{\ell n}$, il peut y avoir un certain nombre de symboles déclarés par déclaration de mode. On peut alors extraire de ℓ une suite qui ne contienne plus ces symboles. Soit ℓ' cette suite :

$$\ell' = \underline{\ell' 1}, \underline{\ell' 2}, \ldots, \underline{\ell' m}, \text{ avec } m \leq n$$

De plus, dans la suite ℓ', il y a en général plusieurs exemplaires d'un même libellé. Ainsi, dans l'exemple précédent, il y a deux exemplaires de var. Soit L l'ensemble des libellés différents contenus dans ℓ' :

$$L = \{\underline{L1}, \underline{L2}, \ldots, \underline{Lp}\}, \text{ avec } p \leq m$$

Dans l'exemple précédent, cet ensemble est :

$$L = \{\underline{var}, \underline{union}, \underline{struct}\}$$

L'ensemble L est appelé l'ensemble caractéristique du circuit. On peut alors donner la définition d'un circuit bien formé.

E.2.2.3. Définition d'un circuit bien formé

Soit L l'ensemble caractéristique d'un circuit contenu dans la représentation temporaire d'un mode. Le circuit est bien formé si et seulement si la proposition suivante est vérifiée :

$$(\underline{var} \in L \vee \underline{seq} \in L) \wedge (\underline{tuple} \in L \vee \underline{struct} \in L)$$

A partir de la définition d'un circuit bien formé, il est maintenant possible de construire la définition complète d'un mode bien formé. Pour cela, on considère seulement les circuits élémentaires, c'est-à-dire ceux qui passent au plus une fois par un même noeud.

E.2.2.4. Définition d'un mode bien formé

Soit C l'ensemble des circuits élémentaires contenus dans la représentation temporaire d'un mode.

- Si C = ∅, ce mode est bien formé.
- Si C ≠ ∅, ce mode est bien formé si et seulement si tous les circuits appartenant à C sont eux-mêmes bien formés.

Ainsi, c'est la notion de circuit bien formé qui est à la base de la notion de mode bien formé. Il est intéressant de noter, à travers quelques exemples, les significations respectives des deux membres de la conjonction qui constitue la proposition définissant ce qu'est un circuit bien formé.

Le premier membre, (var ∈ L ∨ seq ∈ L) sert à éliminer des déclarations comme :

m rep struct (ent a, m b)

c'est-à-dire les déclarations construisant des modes qui correspondraient à des valeurs d'encombrement infini.

Le deuxième membre, (tuple ∈ L ∨ struct ∈ L) sert à éliminer des déclarations comme :

m1 rep var m1
ou m2 rep seq m2
ou m3 rep proc (var m3) n
ou m4 rep proc (n) seq m4

c'est-à-dire des déclarations dont l'utilité est douteuse et la signification obscure. Chacun des deux membres élimine les déclarations sans signification, du genre :

m rep m

Par contre, il est permis d'écrire :

m1 rep struct (ent a, seq m1 b)
et m2 rep tuple (var m2, var m2)

En termes d'implémentations les valeurs qui ont de tels modes occupent un espace fini, si l'on considère que des vlaeurs de mode var m ou seq m peuvent être représentées par des adresses, éventuellement accompagnées de quelques autres informations dont l'encombrement est fixe et indépendant de l'encombrement des valeurs de mode m.

E.2.3. Egalité de deux modes

D'abord étudié par Koster [Ko69], le problème de l'égalité des modes a conduit à la construction de plusieurs algorithmes pour les modes d'Algol 68. La définition de l'égalité de deux modes qui est donnée ici pour BASEL est également bâtie sur un algorithme, qui utilise la représentation définitive des modes définie au paragraphe précédent.

Afin de simplifier un peu cette définition et celle, qui est donnée plus loin, de la relation de compatibilité entre deux modes, il faut d'abord faire quelques remarques sur le mode union.

Dans un mode union, la liste des modes exprime le fait que les objets qui ont ce mode union peuvent prendre des valeurs de l'un quelconque des modes de la liste. Or, syntaxiquement, n'importe quel mode peut être écrit dans cette liste. En particulier, il peut y avoir un autre mode union. Par exemple, on peut construire le mode suivant :

union (ent, bool, union (reel, car))

Un tel mode est celui d'objets de mode union qui peuvent prendre comme valeur des entiers, des booléens ou les valeurs que peuvent aussi prendre les objets de mode union (reel, car), c'est-à-dire des réels ou des caractères. Ainsi, on voit que les mêmes possibilités seront exprimées, plus clairement et plus simplement, par le mode :

union (ent, bool, reel, car)

Ceci peut s'exprimer de façon générale. Soit n le mode union suivant :

union (m1, m2, ..., mi-1, mi, mi+1, ..., mn)

Pour tout i ($1 \leq i \leq n$) tel que mi = union (mi1, mi2, ..., mip), m peut être simplifié et mis sous la forme :

union (m1, m2, ..., mi-1, mi1, mi2, ..., mip, mi+1, ..., mn)

Lorsque le mode union m a ainsi été simplifié et qu'il ne reste plus d'autres modes union dans sa liste de modes, on dit que m a été nivelé.

(D'autre part, il est clair que, dans la liste des modes d'un mode union, l'ordre est sans signification et la répétition d'un mode est identique à un seul exemplaire de ce mode. Ceci est exprimé formellement plus loin dans la définition de l' égalité entre deux modes).

Pour comparer deux modes m et n on comparera leurs représentations définitives, en supposant, ce qui est toujours possible, que tous les modes union qui y sont contenus ont été nivelés.

E.2.3.1. Définition du problème

Si l'on ignore, dans une première approximation, le cas des modes récursifs, tout le processus de comparaison se réduit à la comparaison de deux arbres, que l'on peut schématiser rapidement dans un tableau. Dans ce tableau :

- m', m1, m2, ..., mp, n', n1, n2, ..., nq sont des modes
- r1, r2, ..., rp, s1, s2, ..., sq sont des identificateurs (deux identificateurs sont égaux s'ils sont composés des mêmes caractères dans le même ordre).
- rien est assimilé à un mode, pour abréger le tableau dans le cas de la comparaison de deux modes de procédures.

Le tableau se lit ligne par ligne :

"si m est rien alors m = n si n est rien,
si m est ent alors m = n si n est ent,
...
si m est union (m1, m2, ..., mp) alors m = n
si n est union (n1, n2, ..., nq) avec la condition :

$$\forall i\ (\exists j\ (mi = nj)) \wedge \forall j\ (\exists i\ (mi = nj))"$$

Le processus de comparaison défini par ce tableau est donc construit de façon récursive et il effectue deux parcours simultanés des arbres qui représentent chacun l'un des modes à comparer.

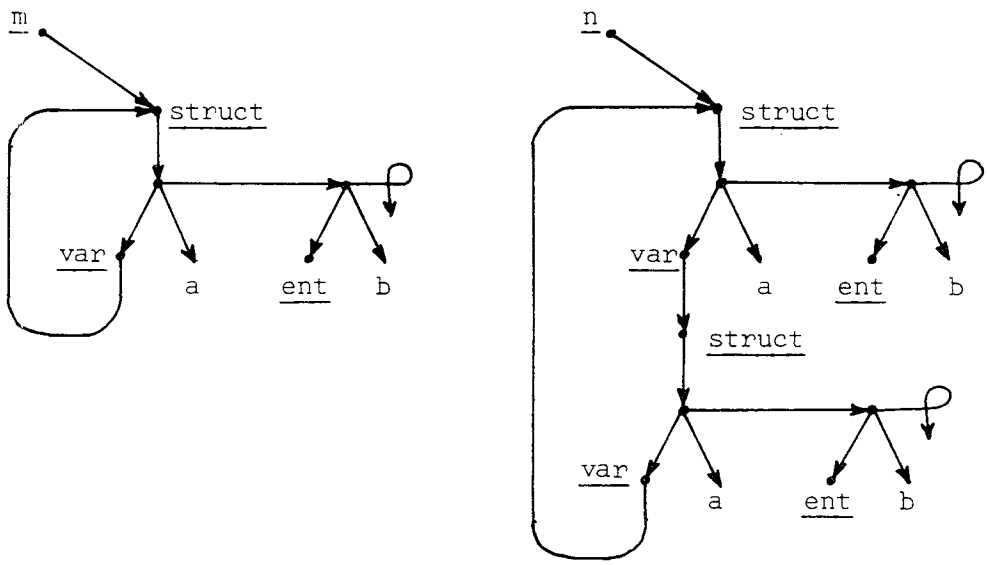
EGALITE DE DEUX MODES NON RECURSIFS

si m est :	alors m = n si n est :
rien	rien
ent	ent
reel	reel
bool	bool
car	car
var m'	var n' avec m' = n'
proc (m1, m2, ..., mp) m'	proc (n1, n2, ..., nq) n' avec m' = n' ∧ p = q ∧ ∀i (mi = ni)
proc m'	proc n' avec m' = n'
tuple (m1, m2, ..., mp)	tuple (n1, n2, ..., nq) avec p = q ∧ ∀i (mi = ni)
struct (m1 r1, m2 r2, ..., mp rp)	struct (n1 s1, n2 s2, ..., nq sq) avec p = q ∧ ∀i (mi = ni ∧ ri = si)
seq m'	seq n' avec m' = n'
union (m1, m2, ..., mp)	union (n1, n2, ..., nq) avec ∀i (∃j (mi = nj)) ∧ ∀j (∃i (mi = nj))

Cependant, cette définition n'est pas suffisante. En effet, soient les modes m et n déclarés par :

m rep struct (var m a, ent b)
n rep struct (var struct (var n a, ent b) a, ent b)

Si l'on veut comparer ces deux modes, le processus décrit par le tableau précédent va "boucler" indéfiniment sur les circuits contenus dans les deux représentations :



Il faut donc pouvoir arrêter le processus par d'autres moyens que la rencontre des "feuilles" de l'arbre. Il faut introduire une information supplémentaire qui permette de dire : "à ce point de la comparaison, on est arrivé sur deux noeuds par lesquels on est déjà passé. Jusqu'à présent, l'égalité a été satisfaite. Si ces noeuds eux-mêmes satisfont toujours l'égalité et si l'on repasse par eux, on vérifiera des égalités que l'on connait déjà. Donc, il n'est pas nécessaire de repasser par ces noeuds. Il suffit alors de les considérer comme des "feuilles" satisfaisant l'égalité et de continuer la comparaison normalement avec cette hypothèse. Si l'égalité doit ne pas être satisfaite, c'est nécessairement dans la suite qu'on le découvrira".

Soient donc deux modes, m et n, éventuellement récursifs, dont on veut savoir s'ils sont égaux. On utilise pour cela les représentations définitives de ces modes, dans lesquelles les modes unions éventuels ont été nivelés.

Soit S un ensemble dont les éléments sont, d'une part les sommets des représentations des modes qui sont des sous-graphes des graphes représentant m et n, et d'autre part le vide, c'est-à-dire le noeud fictif auquel arrivent les arcs qui "vont vers le vide".

Soit M un élément de S. On définit alors une fonction L qui à M fait correspondre soit le libellé qu'il porte si M est le sommet de la représentation d'un mode, soit un symbole spécial, nil, si M est le vide :

$$L(M) \in \{\underline{ent}, \underline{reel}, \underline{bool}, \underline{car}, \underline{var}, \underline{proc}, \underline{tuple}, \underline{struct}, \underline{seq}, \underline{union}, \underline{nil}\}$$

A partir de la définition de L(M), on peut définir une partition de S, en deux parties appelées NT et T, pour Non Terminal et Terminal respectivement :

$$NT = \{M \in S \mid L(N) \in \{\underline{var}, \underline{proc}, \underline{tuple}, \underline{struct}, \underline{seq}, \underline{union}\}\}$$

$$T = \{M \in S \mid L(M) \in \{\underline{ent}, \underline{reel}, \underline{bool}, \underline{car}, \underline{nil}\}\}$$

On définit alors, sur NT, une fonction K qui à M fait correspondre un entier naturel :

$$K(M) \in N$$

Cet entier est appelé la classe de M. Comme on le verra par la suite, c'est la classe d'un noeud qui constitue cette information supplémentaire dont on a besoin pour décider des points d'arrêts lors du parcours des graphes dans le cas de modes récursifs. La classe de chaque $M \in NT$ sera déterminée par le processus de comparaison lui-même. Avant que ne commence ce processus, $K(M) = 0$ pour tous les éléments M de NT.

Si C est une classe, la définition du processus de comparaison l'attribue à un noeud M par la notation :

$$K(M) \leftarrow C$$

Soit enfin I une variable à valeurs dans N, dont le processus de comparaison se sert pour créer de nouvelles classes :

- Avant que recommence le processus de comparaison, $I = 0$.
- Soit k la valeur de I à un instant donné de la comparaison. Si le processus de comparaison demande alors de créer une nouvelle classe, demande exprimée par la notation NOUVELLE CLASSE, I prend la valeur $k + 1$.

On définit également plusieurs fonctions auxiliaires qui servent à parcourir les graphes qui représentent les modes.

(Remarque : on aurait pu, pour la représentation des modes, utiliser le formalisme de Vienne. Dans ce cas, ces fonctions auxiliaires seraient des "sélecteurs". Le formalisme d'AMBIT/G, introduit plus loin, permet de décrire n'importe quelle sorte de graphe orienté et utilise lui aussi, sous une forme graphique, la notion de sélecteur).

Soit M ∈ NT :

- Si L(M) = var ou si L(M) = seq, la fonction VAL(M) a pour valeur le sommet de la représentation du mode qui est à l'extrémité de l'arc issu de M.
- Si L(M) = proc, la fonction PARAM (M) a pour valeur le sommet de la représentation de la liste des modes des paramètres, ou le vide s'il n'y a pas de paramètre. La fonction RES(M) a pour valeur le sommet de la représentation du mode du résultat, ou le vide s'il n'y a pas de résultat.
- Si L(M) = tuple, la fonction ELEM(M) a pour valeur le sommet de la représentation de la liste des modes des éléments.
- Si L(M) = struct, la fonction CHAMP(M) a pour valeur le sommet de la représentation de la liste des champs.
- Si L(M) = union, la fonction POSSIB(M) a pour valeur le sommet de la représentation de la liste des modes possibles.
- Soit P le sommet de la représentation d'une liste de modes :
  - . NB(P) est le nombre de modes de cette liste.
  - . Si k est un entier ($1 \leq k \leq NB(P)$), MODE(P, k) est le sommet de la représentation du $k^{ème}$ mode de la liste.
- Soit Q le sommet de la représentation d'une liste de champs :
  - . NB(Q) est le nombre de champs de cette liste.
  - . Si k est un entier ($1 \leq k \leq NB(Q)$), MODE(Q, k) est le sommet de la représentation du mode du $k^{ème}$ champs de la liste et SELECT(Q, k) est le sélecteur de ce champ. (Comme il est sans intérêt de détailler ici la représentation des sélecteurs, il suffit de dire que deux sélecteurs sont égaux quand ils sont composés des mêmes caractères dans le même ordre).

E.2.3.2. Processus de comparaison de deux modes

Le processus de comparaison est décrit ici de façon algorithmique, par un ensemble de fonctions à valeurs booléennes qui s'appellent de façon récursives. Chaque fonction est décrite sous la forme générale suivante :

```
F (M, N) <= {C1 → E1•
             C2 → E2•
              ...
             Cn → En•
             E n+1}
```

où : F est le nom de la fonction,
M et N sont ses paramètres,
Ci est une condition,
Ei est une expression ou, entre parenthèses, une série d'expressions séparées par des points-virgules.

La syntaxe des conditions et des expressions étant très simple, elle ne requiert pas de description plus détaillée.

Les Ci → Ei sont les éléments du corps de la fonction, qui se comporte comme une expression conditionnelle généralisée : les Ci sont examinés dans l'ordre de leur écriture. Dès que l'un d'entre eux est satisfait, c'est le calcul du Ei correspondant qui détermine la valeur de F. Si aucun des Ci n'est satisfait, c'est le calcul de E n+1 qui détermine la valeur de F.

Ces définitions préliminaires étant faites, soient m et n les deux modes à comparer, et soient M et N les sommets des représentations respectives de m et n. Si la fonction EGAL, définie ici, a pour valeur vrai, m et n sont égaux. Si elle a pour valeur faux, m et n ne sont pas égaux.

```
EGAL (M, N) <= {L(M) ∈ T ∧ L(N) ∈ T ∧ L(M) = L(N) → vrai•
                L(M) = var ∧ L(N) = var → VAREGAL(M, N)•
                L(M) = proc ∧ L(N) = proc → PROEGAL(M, N)•
                L(M) = tuple ∧ L(N) = tuple → TUPEGAL(M, N)•
                L(M) = struct ∧ L(N) = struct → STREGAL(M, N)•
                L(M) = seq ∧ L(N) = seq → SEQEGAL(M, N)•
                L(M) = union ∧ L(N) = union → UNIEGAL(M, N)•
                faux}
```

Les fonctions auxiliaires VAREGAL, PROEGAL, TUPEGAL, STREGAL, SEQEGAL et UNIEGAL ont toutes la même structure. Elles diffèrent seulement par une expression. Soit E cette expression et F l'une quelconque de ces six fonctions auxiliaires. Leur définition générale peut alors être donnée sous la forme suivante :

```
F(M, N) <= {K(M) ≠ 0 ∧ K(N) ≠ 0 ∧ K(M) = K(N) → vrai•
            K(M) = 0 ∧ K(N) = 0 → (NOUVELLE CLASSE ;
                                   K(M) ← I ;
                                   K(N) ← I ;
                                   E)•
            K(M) ≠ 0 ∧ K(N) = 0 → (K(N) ← K(M) ; E)•
            K(M) = 0 ∧ K(N) ≠ 0 → (K(M) ← K(N) ; E)•
            faux}
```

Pour F = VAREGAL, E = EGAL(VAL(M), VAL(N))•

Pour F = PROEGAL, E = LISTMEGAL(PARAM(M), PARAM(N))
∧ EGAL(RES(M), RES(N))•

Pour F = TUPEGAL, E = LISTMEGAL(ELEM(M), ELEM(N))•

Pour F = STREGAL, E = LISTCEGAL(CHAMP(M), CHAMP(N))•

Pour F = SEQEGAL, E = EGAL(VAL(M), VAL(N))•

Pour F = UNIEGAL, E = ENSMEGAL(POSSIB(M), POSSIB(N))•

Enfin, il reste à définir trois fontions :

- LISTMEGAL, qui permet de comparer deux ensembles ordonnés de modes.
- LISTCEGAL, qui permet de comparer deux ensembles ordonnés de champs.
- ENSMEGAL, qui permet de comparer deux ensembles non ordonnés de modes.

Dans les définitions de ces fonctions il est fait usage des quantifieurs universel et existentiel, selon les conventions suivantes :

- l'expression $\forall k \in [1, p]$ (E) a pour valeur vrai si et seulement si pour tout entier k, tel que $1 \leq k \leq p$, l'expression E a la valeur vrai.
- l'expression $\exists k \in [1, p]$ (E) a pour valeur vrai si et seulement si il existe au moins une valeur de k, telle que $1 \leq k \leq p$, pour laquelle l'expression E a la valeur vrai.

Les définitions des trois dernières fonctions sont alors :

```
LISTMEGAL(M, N) <= {NB(M) = NB(N)
                    ∧ ∀k ∈ [1, NB(M)] (EGAL(MODE(M, k), MODE(N, k)))}

LISTCEGAL(M, N) <= {NB(M) = NB(N)
                    ∧ ∀k ∈ [1, NB(M)] (EGAL(MODE(M, k), MODE(N, k))
                                       ∧ SELECT(M, k) = SELECT(N, k))}

ENSMEGAL(M, N)  <= {∀k ∈ [1, NB(M)]
                                (∃ℓ ∈ [1, NB(N)] (EGAL(MODE(M, k), MODE(N, ℓ))))
                    ∧ ∀ℓ ∈ [1, NB(N)]
                                (∃k ∈ [1, NB(M)] (EGAL(MODE(M, k), MODE(N, ℓ))))}
```

Cette définition algorithmique de l'égalité entre deux modes est donc structurée en trois niveaux :

1 - La fonction EGAL, qui compare les noeuds terminaux des représentations, ou qui aiguille vers les fonctions du deuxième niveau si les noeuds à comparer ne sont pas terminaux.

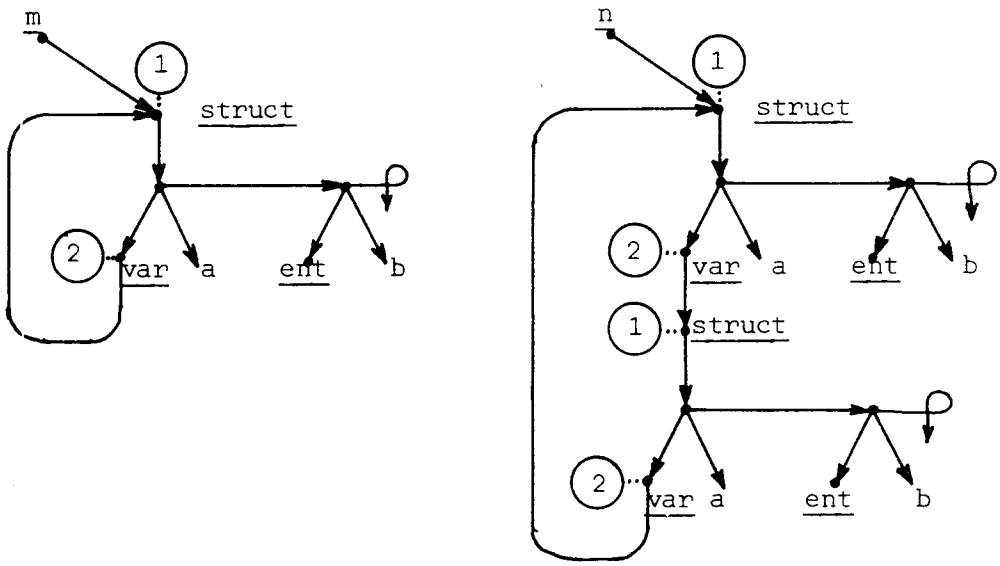
2 - les fonctions VAREGAL, PROEGAL, TUPEGAL, STREGAL, SEQEGAL et UNIEGAL, qui déterminent et comparent les classes des noeuds non terminaux :

   . Si les classes sont déjà déterminées et égales, il n'est plus nécessaire de repasser par ces noeuds et l'égalité est satisfaite.
   . Si les classes de deux noeuds n'ont pas encore été déterminées, on crée une nouvelle classe qu'on leur attribue et on passe par ces noeuds.
   . Si la classe d'un seul des noeuds n'est pas encore déterminée, on lui attribue celle de l'autre, et on passe par ces deux noeuds.

. Si les classes des deux noeuds sont déterminées et différentes, l'égalité n'est pas satisfaite.

3 - les fonctions LISTMEGAL, LISTCEGAL et ENSMEGAL, qui sont de simples fonctions de service pour effectuer des itérations sur des ensembles.

Ainsi, lorsque la comparaison est terminée, et que le résultat est vrai, tous les noeuds qui ont la même classe sont les sommets de modes égaux entre eux. Par exemple, après avoir comparé les modes m et n déclarés plus haut, les classes des noeuds seront les suivantes :



Ces deux modes sont donc égaux, et n contient deux représentations d'un mode égal à lui-même.

E.2.4. Compatibilité d'un mode avec un autre

En BASEL, comme en Algol 68, il y a un mécanisme de modification. Si une valeur de mode m est obtenue dans un contexte qui exige une valeur de mode n, deux cas sont possibles :

- m = n, auquel cas il n'y a aucune modification à effectuer sur la valeur de mode m.

- m ≠ n, auquel cas il y a lieu, si cela est possible, d'effectuer sur la valeur de mode m un certain nombre de modifications qui la transformeront en une valeur de mode n, ces modifications n'ayant pas à être exprimées explicitement dans le programme. Il s'agit donc, comme en Algol 68, d'une généralisation du mécanisme de conversion automatique existant dans d'autres langages.

Si m = n, ou s'il existe une combinaison de modifications qui permet de transformer une valeur de mode m en une valeur de mode n, on dit que m est compatible avec n, ce qui est noté :

$$m \Rightarrow n$$

Cela signifie donc que là où une valeur de mode n est exigée, il est possible d'écrire une expression à résultat de mode m. La définition de cette relation de compatibilité peut être donnée sous la forme d'un tableau, qui se lit ligne par ligne comme celui définissant l'égalité entre deux modes non récursifs.

Dans ce tableau :

- m et n sont les modes à propos desquels on veut savoir si m ⇒ n.
- m1, m2, ..., mp, n1, n2, ..., nq sont des modes.
- m' est n' sont des modes, ou rien.
- r1, r2, ..., rp, s1, s2, ..., sq sont des identificateurs.

La définition de la relation de compatibilité est construite de façon récursive à partir de deux parcours simultanés des graphes qui constituent les représentations définitives respectives de m et n. C'est la relation d'égalité qui sert de point d'arrêt dans le fonctionnement récursif de cette définition. En effet, dès qu'on est parvenu, en appliquant une composition de modifications, à transformer le mode m en un mode égal à n, on sait que m ⇒ n.

Chaque ligne de cette définition correspond à un genre particulier de modification :

ligne (1) : pas de modification, c'est l'égalité.

lignes (2) et (3) : modification élémentaire appelée réunion.

ligne (4) : modification élémentaire appelée extraction de valeur d'une variable.

ligne (5) : modification élémentaire appelée appel implicite de procédure.

ligne (6) : modification composée appelée modification de tuple.

ligne (7) : modification composée appelée structuration de tuple.

ligne (8) : modification composée appelée mise en séquence d'un tuple.

ligne (9) : modification composée appelée modification de structure.

COMPATIBILITE D'UN MODE AVEC UN AUTRE

si m est :	alors m ⇒ n si n est :
(1) m'	n' avec m' = n'
(2) m'	union (n1, n2, ..., nq) avec ∃i (m' = ni)
(3) union (m1, m2, ..., mp)	union (n1, n2, ..., nq) avec ∀i (∃j (mi = nj))
(4) var m'	n' avec m' ⇒ n'
(5) proc m'	n' avec m' ⇒ n'
(6) tuple (m1, m2, ..., mp)	tuple (n1, n2, ..., nq) avec p = q ∧ ∀i (mi ⇒ ni)
(7) tuple (m1, m2, ..., mp)	struct (n1 s1, n2 s2, ..., nq sq) avec p = q ∧ ∀i (mi ⇒ ni)
(8) tuple (m1, m2, ..., mp)	seq n' avec ∀i (mi ⇒ n')
(9) struct (m1 r1, m2 r2, ..., mp rp)	struct (n1 s1, n2 s2, ..., nq sq) avec p = q ∧ ∀i (mi ⇒ ni ∧ ri = si)

La sémantique dynamique de ces diverses modifications est décrite au paragraphe L sur les modifications explicites.

D'autre part, une modification composée, appelée suppression de valeur n'entre pas dans ce tableau à cause de sa nature particulière. Elle est étudiée au paragraphe F.2.1.2. à propos de la description des parties instructions.

E.2.5. Commentaires

Ces longues définitions relatives au traitement statique des modes montrent l'importance de la notion de mode en BASEL. Comme Algol 68, c'est un langage essentiellement construit autour de cette notion. On remarque cependant que tous les modes de BASEL sont "virtuels", au sens que le rapport Algol 68 donne à ce terme.

La représentation des modes sous forme de graphe, qui sert de support à la plupart des définitions précédentes, est très proche de celle qu'utiliserait un compilateur. Ceci se vérifie d'ailleurs non seulement dans les deux implémentations de BASEL, mais aussi dans sa définition formelle en AMBIT/G.

L'algorithme de comparaison de deux modes est également celui que peut utiliser un compilateur. D'ailleurs, cet algorithme a comme avantage que lorsque deux modes ont été trouvés égaux, on a obtenu, en "sous-produit", des classes de modes égaux. Ceci peut être avantageusement utilisé dans la construction d'une table de modes par le compilateur, dans laquelle chaque mode n'est représenté qu'une fois et, si possible, sous sa forme la plus condensée lorsqu'il s'agit d'une union ou d'un mode récursif. On peut remarquer à ce propos que Madame BAJAR a proposé dans sa thèse un premier algorithme, légèrement différent de celui-ci, qui sert à comparer deux modes, et un deuxième algorithme qui, étant donné un mode, en construit la représentation la plus condensée. D'autre part, en associant un entier à chaque mode différent des autres, l'algorithme donné ici crée la représentation la plus simple que l'on peut imaginer pour représenter dans un objet de mode union le mode de la valeur qu'il a prise à un instant donné.

Enfin, la définition de la relation de compatibilité regroupe de façon concise tous les cas possibles de modification (à l'exclusion du cas particulier de la suppression de valeur). Dans les paragraphes qui suivent, les divers contextes où ces modifications peut intervenir sont décrits, ainsi que les diverses restrictions auxquelles leur composition peut être soumise dans certains contextes.

F. Les instructions

F.1. Syntaxe

partie-instruction : branche-d-instructions <| branche-d-instructions> ...
branche-d-instructions : séquence-d-instructions <→ séquence-d-instructions> ...
sequence-d-instructions : instruction <; instruction> ...
instruction : <identificateur :> ... <allera|expression>
allera : allera identificateur

EXEMPLE :

inf (a, 0) → sous (0, a) | a

F.2. Sémantique

F.2.1. Statique

Dans ce paragraphe et dans les suivants, une part importante de la sémantique statique concerne la façon dont sont appliquées et composées les modifications. Il convient donc de commencer par introduire ici les conventions qui sont utilisées à ce propos.

F.2.1.1. Mise en oeuvre des modifications

Soit I une instruction, une expression, ou l'une quelconque des formes syntaxiques que peut prendre une instruction. On suppose que I est à résultat de mode m et que cette instruction est écrite dans un contexte qui exige une instruction à résultat de mode m'.

Si m = m', il n'y a aucune modification à appliquer.

Si m ≠ m', et s'il n'est pas vrai que m ⇒ m', le programme est erroné.

Si m ≠ m', avec m ⇒ m', il existe une combinaison de modifications qui, dynamiquement, transformera le résultat de mode m en un résultat du mode m' demandé. Cette combinaison de modifications est déterminée statiquement et elle a pour effet statique une transformation de l'instruction I en une nouvelle instruction à résultat de mode m' qui vient prendre la place de I dans le programme. Avec m ≠ m' et m ⇒ m', cette transformation sera notée conventionnellement :

$$I \leftarrow f(m', I)$$

où f est différent selon la modification qui lui correspond.

Les diverses formes de f sont, pour les sept modifications déjà définies :

- ex(m', I) pour l'extraction de valeur (m = var m') ;
- ap(m', I) pour l'appel implicite (m = proc m') ;
- un(m', I) pour la réunion (m' = union (m'1, m'2, ..., m'n),
  m ≠ m',
  m ⇒ m') ;
- tt(m', I) pour la modification de tuple (m' = tuple(m'1, m'2, ..., m'n),
  m = tuple(m1, m2, ..., mn),
  m ≠ m',
  m ⇒ m') ;
- ts(m', I) pour la structuration (m' = struct(m'1 s1, m'2 s2, ..., m'n sn),
  m = tuple(m1, m2, ..., mn),
  m ⇒ m') ;
- tq(m', I) pour la mise en séquence (m' = seq m",
  m = tuple(m1, m2, ..., mn),
  m ⇒ m') ;
- ss(m', I) pour la modification de structure
  (m' = struct(m'1 s1, m'2 s2, ..., m'n sn),
  m = struct(m1 s1, m2 s2, ..., mn sn),
  m ≠ m',
  m ⇒ m')

En utilisant ces diverses formes de f, on définit également des "fonctions" logiques. Ce sont ces fonctions qui seront utilisées par la suite dans la sémantique statique pour exprimer les conditions d'application de la relation de compatibilité et les modifications qui sont autorisées dans les divers contextes d'une partie instruction. Chacune de ces fonctions a pour valeur soit VRAI soit FAUX, et quand sa valeur est VRAI avec les relations $m \neq m'$ et $m \Rightarrow m'$, elle applique à I la transformation appropriée.

Ces fonctions logiques sont les suivantes :

EX(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m = var m', auquel cas I subit la transformation :
I ← ex(m', I).
- FAUX autrement.

AP(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m = proc m', auquel cas I subit la transformation :
I ← ap(m', I).
- FAUX autrement.

UN(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m' = union(m'1, m'2, ..., m'n)
∧ m ≠ m'
∧ m ⇒ m'
auquel cas I subit la transformation :
I ← un(m', I).
- FAUX autrement.

TT(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m' = tuple(m'1, m'2, ..., m'n)
∧ m = tuple(m1, m2, ..., mn)
∧ m ≠ m'
∧ m ⇒ m'
auquel cas I subit la transformation :
I ← tt(m', I).
- FAUX autrement.

TS(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m' = struct(m'1 s1, m'2 s2, ..., m'n sn)
∧ m = tuple(m1, m2, ..., mn)
∧ m ⇒ m'
auquel cas I subit la transformation :
I ← ts(m', I).
- FAUX autrement.

TQ(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m' = seq m"
∧ m = tuple(m1, m2, ..., mn)
∧ m ⇒ m'
auquel cas I subit la transformation :
I ← tq(m', I).
- FAUX autrement.

SS(m', I) = - VRAI si m = m', auquel cas I est inchangé.
- VRAI si m' = struct(m'1 s1, m'2 s2, ..., m'n sn)
∧ m = struct(m1 s1, m2 s2, ..., mn sn)
∧ m ≠ m'
∧ m ⇒ m'
auquel cas I subit la transformation :
I ← ss(m', I).
- FAUX autrement.

D'autre part, la définition de la relation de compatibilité entre deux modes fait apparaître que les modifications peuvent être composées. Ainsi, si I a un résultat de mode m = var proc var ent et si le contexte exige m' = ent , I subira la composition de transformations suivante pour traduire les modifications qui seront appliquées à son résultat :

I ← ex(ent , ap(var ent, ex(proc var ent, I)))

Ainsi, si m ⇒ m', on notera, d'une façon générale, f une composition quelconque de transformations traduisant les modifications nécessaires pour obtenir une valeur de mode m' à partir d'une valeur de mode m. Cette transformation générale est donc définie comme suit :

- I ← f(m', I) ne change pas I si m = m'.
- I ← f(m', I) applique à I les transformations nécessaires si m ≠ m'.

A f est associée une fonction logique F :

F(m', I) = - VRAI si m ⇒ n', auquel cas I subit la transformation :

$$I \leftarrow f(m', I)$$

- FAUX autrement.

Etant donné un mode m et un mode m', la combinaison de modifications qui permet de "passer" de m à m' est toujours unique, en raison de la façon dont est définie la relation de compatibilité entre deux modes. Ceci a été démontré par Madame V. BAJAR dans sa thèse [Ba73].

Enfin, trois compositions de transformation méritent d'être distinguées car elles sont fréquemment utilisées par la suite.

- Soit $\underline{var}^n$, n ≥ 1, une suite de n symboles var et $\underline{var}^*$ une suite de p, p ≥ 0, symboles var.
- Soit $\underline{proc}^n$, n ≥ 1, une suite de n symboles proc et $\underline{proc}^*$ une suite de p, p ≥ 0, symboles proc.
- Soit (var + proc) une notation conventionnelle pour représenter indifféremment un symbole var ou un symbole proc.
- Soit $(\underline{var} + \underline{proc})^n$, n ≥ 1, une suite de n symboles (var + proc) et $(\underline{var} + \underline{proc})^*$ une suite de p, p ≥ 0, symboles (var + proc).
- Soit exap une notation conventionnelle pour représenter indifféremment ex ou ap.

On définit donc, en supposant toujours que m ⇒ m' :

$I \leftarrow ex^*(m', I)$, avec la signification :

- si m = m', I est inchangé.
- si $m = \underline{var}^n\ m'$, I subit la transformation :

$$I \leftarrow ex(m', ex(\underline{var}\ m', \ldots, ex(\underline{var}^{n-1}\ m', I) \ldots))$$

$I \leftarrow ap^*(m', I)$, avec la signification :

- si m = m', I est inchangé.
- si $m = \underline{proc}^n\ m'$, I subit la transformation :

$$I \leftarrow ap(m', ap(\underline{proc}\ m', \ldots, ap(\underline{proc}^{n-1}\ m', I) \ldots)).$$

$I \leftarrow (ex + ap)^*(m', I)$, avec la signification :

- si $m = m'$, I est inchangé.
- si $m = (\underline{var} + \underline{proc})^n\ m'$, I subit la transformation :

$$I \leftarrow exap(m', exap((\underline{var} + \underline{proc})\ m', \ldots, exap((\underline{var} + \underline{proc})^{n-1}\ m', I) \ldots)).$$

A ces trois transformations composées correspondent trois fonctions logiques :

$EX^*(m', I)$ = - VRAI si $m = \underline{var}^* m'$, auquel cas I subit la transformation :

$$I \leftarrow ex^*(m', I)$$

- FAUX autrement.

$AP^*(m', I)$ = - VRAI si $m = \underline{proc}^* m'$, auquel cas I subit la transformation :

$$I \leftarrow ap^*(m', I)$$

- FAUX autrement.

$(EX + AP)^*(m', I)$ = - VRAI si $m = (\underline{var} + \underline{proc})^* m'$, auquel cas I subit la transformation :

$$I \leftarrow (ex + ap)^*(m', I)$$

- FAUX autrement.

F.2.1.2. La suppression de valeur

La dernière modification, appelée suppression de valeur, a pour rôle de transformer une instruction I à résultat de mode m en une instruction à résultat vide. (Le vide, qui est l'absence de valeur, n'a pas de mode : c'est pourquoi cette modification n'a pas été introduite à l'occasion de la définition de la relation de compatibilité entre deux modes).

On note la transformation statique correspondante de la façon suivante :

$$I \leftarrow sv(I)$$

A cette transformation correspond une fonction logique, définie comme suit :

SV(I) = - VRAI si I est à résultat vide, auquel cas I est inchangé.
- VRAI si I est une instruction allera, auquel cas I est inchangé.
- VRAI si I est une association, auquel cas I est transformé par :

$$I \leftarrow sv(I).$$

- VRAI si $m \neq (\underline{var} + \underline{proc})^* \underline{proc}\ m'$
$\wedge$ (m' = rien $\vee$ m' = un mode quelconque),
auquel cas I est transformé par :

$$I \leftarrow sv(I).$$

- VRAI si $m = (\underline{var} + \underline{proc})^* \underline{proc}\ \underline{rien}$, auquel cas I est transformé par :

$$I \leftarrow ap(\underline{rien}, (ex + ap)^*(\underline{proc}\ \underline{rien}, I)).$$

- VRAI si $m = (\underline{var} + \underline{proc})^* \underline{proc}\ m'$
$\wedge$ $m' \neq \underline{rien}$
$\wedge$ $m' \neq (\underline{var} + \underline{proc})^* m''$
$\wedge$ (m'' = rien $\vee$ m'' = un mode quelconque),
auquel cas I est transformé par :

$$I \leftarrow sv(ap(m', (ex + ap)^*(\underline{proc}\ m', I))).$$

On remarque donc que la fonction SV(I) est identiquement VRAIe mais qu'elle provoque des transformations différentes selon le mode m du résultat de I.

F.2.1.3. Sémantique statique de la partie instruction

Soit b1, b2, ..., bn la liste, dans l'ordre, des branches d'instructions d'une partie instruction. Si bn est composé de plusieurs séquences d'instruction, c'est-à-dire s'il a la forme :

$$s1 \rightarrow s2 \rightarrow \ldots \rightarrow sq$$

où les si sont des séquences d'instructions, l'instruction vide est rajoutée pour constituer une $n+1^{ème}$ branche d'instructions, la partie instruction prenant alors la configuration :

$$b1 \mid b2 \mid \ldots \mid bn \mid \underline{vide}$$

Parmi les branches d'instructions de la partie instruction, incluant eventuellement la branche bn+1 (c'est-à-dire vide) rajoutée après bn, soient bi1, bi2, ..., bip celles qui ne sont pas "sans issue" (voir plus bas la définition de ce terme). Ce sont ces p branches d'instructions que l'on considère pour déterminer le mode du résultat de la partie instruction. Soit Ij ($j \in \{i1, i2, \ldots, ip\}$) la dernière instruction de la dernière séquence d'instructions de la branche d'instructions bij.

- Si p = 0, le programme est erroné, car il ne se termine pas.
- Si p ≥ 1, et s'il y a au moins une branche d'instructions à résultat vide, on effectue :

$$\bigwedge_{j \in \{i1, i2, \ldots, ip\}} (SV(I_j))$$

  ce qui est identiquement VRAI.
- Si p ≥ 1, et s'il n'y a pas de branche d'instructions à résultat vide, soient alors mi1, mi2, ..., mip les modes respectifs des résultats des p branches d'instructions considérées.

Deux cas se présentent alors :

- Si mi1 = mi2 = ... = mip, le mode du résultat de la partie instruction est le mode commun des résultats des p branches d'instructions considérées.
- S'il existe au moins deux indices ik et iℓ tels que mik ≠ miℓ, alors on effectue :

$$\bigwedge_{j \in \{i1, i2, \ldots, ip\}} (UN(\underline{union}(mi1, mi2, \ldots, mip), Ij))$$

  ce qui, dans ce cas, est toujours VRAI. Le mode du résultat de la partie instruction est union(mi1, mi2, ..., mip).

Soit s1, s2, ..., sℓ la liste, dans l'ordre, des séquences d'instructions d'une branche d'instructions. Soit J_j ($1 \leq j \leq \ell-1$) la dernière instruction de la séquence sj.

- Si sℓ est sans issue, la branche d'instructions est sans issue.
- Si sℓ est à résultat vide, la branche d'instructions est à résultat vide.
- Si sℓ est à résultat de mode m, la branche d'instructions est à résultat de mode m.

On effectue :

$$\bigwedge_{i \le j \le \ell-1} ((EX + AP)^*(\underline{bool}, J_j))$$

qui doit être VRAI pour que le programme soit correct.

Soit e1, e2, ..., ek la liste, dans l'ordre, des instructions d'une séquence d'instructions.

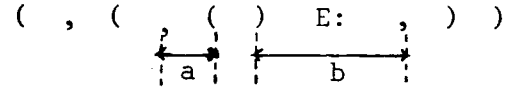
- Si ek est une instruction allera, la séquence d'instructions est sans issue.
- Si ek est à résultat vide, la séquence d'instructions est à résultat vide.
- Si ek est à résultat de mode m, la séquence d'instructions est à résultat de mode m.

On effectue :

$$\bigwedge_{i \le j \le k-1} (SV(ej))$$

ce qui est identiquement VRAI.

Les identificateurs suivis de ":" qui apparaissent éventuellement au début d'une instruction définissent des étiquettes pour cette instruction. Ces étiquettes, contrairement aux règles en vigueur dans la plupart des langages classiques à structure de bloc, ont une portée limitée au seul niveau de bloc, ou de texte de procédure, où elles sont définies, à l'exclusion des blocs plus intérieurs qui peuvent y être inclus. Ainsi dans :



si les parenthèses délimitent des expression composées et les virgules des blocs, l'étiquette E n'est "connue" que dans les parties a et b.

L'identificateur utilisé dans une instruction allera doit être une étiquette, dont la définition doit donc être faite au même niveau de bloc que celui où cette instruction est écrite. Ainsi, une instruction allera ne peut en aucune façon renvoyer à un niveau de bloc autre que le sien propre.

Dynamique

L'exécution d'une partie instruction commence par l'éxécution de la première instruction de la première séquence d'instructions de sa première branche d'instructions.

Les instructions d'une séquence d'instructions sont exécutées les unes après les autres, de façon sérielle, l'instruction allera ayant la signification habituelle que lui ont attribuée bien d'autres langages.

Une séquence d'instructions qui n'est pas sans issue est terminée quand sa dernière instruction a été exécutée. La valeur d'une séquence d'instructions est celle, si elle existe, de sa dernière instruction, valeur qui n'est obtenue qu'en cas de terminaison de la séquence d'instructions. Soit une séquence d'instructions suivie d'un signe → :

- Si sa valeur est vrai, alors c'est la séquence d'instructions qui suit immédiatement le signe →, dont l'exécution est commencée à partir de sa première instruction.
- Si sa valeur est faux, alors c'est la branche d'instructions qui suit le prochain signe |, dont l'exécution est commencée, à partir de la première instruction de sa première séquence d'instructions.

Une branche d'instructions qui n'est pas sans issue est terminée quand sa dernière séquence d'instructions est terminée. La valeur d'une branche d'instructions est celle, si elle existe, de sa dernière séquence d'instructions, valeur qui n'est obtenue qu'en cas de terminaison de la branche d'instructions.

Une partie instruction est terminée dès que l'une de ses branches d'instructions est terminée. Sa valeur est alors celle, si elle existe, de la branche d'instructions qui a provoqué sa terminaison.

Commentaires

La structure d'une partie instruction est celle d'une expression conditionnelle généralisée. Ainsi, on peut écrire une expression composée ne comportant qu'un bloc, éventuellement sans déclarations, de la façon suivante :

```
(b1 → e1 |
    b2 → e2 |
        b3 → e3|e4)
```

ceci signifie :

si b1 alors e1 sinsi b2 alors e2 sinsi b3 alors e3 sinon e4 fsi

Les conditions usuelles de la forme si b alors e1 sinon e2 fsi ne sont donc qu'un cas particulier de la structure générale des expressions composées de BASEL :

```
(b → e1 | e2)
```

Une telle organisation a l'avantage de regrouper dans une notation uniforme et avec un mécanisme unique le sinon, le sinsi et le exit d'Algol 68, car, si Iij est une instruction et Ei une étiquette, on peut également écrire :

```
( ... déclarations ... dans
            Io1 ; Io2 ; ... ; Iom|
        E1 : I11 ; I12 ; ... ; I1n|
        E2 :     ...              |
        .
        .
        .
        Eq : Iq1 ; Iq2 ; ... ; Iqp)
```

On remarque d'autre part qu'un souci de limpidité du mécanisme des modifications (il ne faut pas oublier que BASEL est un langage de base) a fait exclure tout mécanisme complexe d'équilibrage : seules la réunion et la suppression de valeur peuvent être appliquées, pour raison d'équilibrage, aux résultats de branches d'instructions.

Enfin, la restriction imposée à l'utilisation de l'instruction allera (elle ne permet pas de "sortir" d'un bloc) l'a été dans le souci de favoriser une structuration claire des programmes, en canalisant sévèrement le passage de l'exécution d'un bloc à un autre :

- on ne peut "entrer" dans un bloc que par son prologue normal ;
- on ne peut "sortir" d'un bloc que par la terminaison de sa partie instruction.

G. Les expressions

G.1. Syntaxe

expression : association|
terme
association : <origine => ... origine = terme
origine : <identificateur de> ... origine-simple
origine-simple : identificateur

EXEMPLES :
Dans la portée des déclarations :

I est ent, J est ent, S est struct (ent a, bool b),
I est tuple(ent, bool),

On peut écrire :

I = 1 ; J = I = 2 ;
S = (3, vrai) ; T = (4, faux) ; S = T ; a de S = 5

G.2. Sémantique

G.2.1. Statique

L'association sert à associer une valeur à une origine.

L'identificateur qui constitue l'origine simple utilisé dans une association doit avoir été déclaré par une déclaration simple, une déclaration conditionnelle ou une déclaration additionnelle rattachée à une procédure

générique. Cette déclaration peut être faite dans le bloc où est écrite l'association, ou dans un bloc l'englobant, à condition que, si l'association est à l'intérieur d'un texte de procédure, la déclaration ne soit pas à l'extérieur de ce texte de procédure. En effet, une procédure ne peut pas faire usage d'une origne identifiée par un identificateur déclaré à son extérieur, mais seulement de la valeur associée à cette origine au moment où son texte de procédure a été évalué (voir le paragraphe M.2.2.2.).

Dans le cas où l'origine écrite dans une association n'est pas une procédure générique, soit I l'identificateur de l'origine simple.

Si le mode spécifié dans la déclaration de I est struct(m1 s1, m2 r2, ..., mp rp), alors il est possible, par utilisation des sélecteurs de champs, d'identifier, dans l'origine simple à laquelle peuvent être associées des valeurs de mode struct(m1 r1, m2 r2, ..., mp rp), l'origine à laquelle peuvent être associées des valeurs de mode mi, en écrivant :

ri de I

Si, à son tour, ri de I identifie une origine à laquelle peuvent être associées des valeurs de mode struct(n1 s1, n2 s2, ..., nq sq), on peut identifier l'origine à laquelle peuvent être associées des valeurs de mode nj, en écrivant :

sj de ri de I

Etc....

L'origine n'étant pas une procédure générique, on connait directement, par la déclaration simple ou conditionnelle, le mode des valeurs qui peuvent lui être associées. Soit alors :

0 = T

une représentation de l'association et m le mode des valeurs qui peuvent être associées à l'origine identifiée par 0. On effectue alors :

F(m, T)

qui doit être VRAI pour que le programme soit correct.

Dans le cas où l'origine est une procédure générique P, l'origine à laquelle une valeur sera associée est identifiée comme suit :

- Soient b1 le bloc où est écrite l'association, b2 le bloc l'englobant, b3 le bloc englobant b2, etc... jusqu'à bn qui est soit le premier corps de procédure rencontré, soit le bloc où est faite la déclaration de générique de P.
- Soient pi1, pi2, ..., piℓ les déclarations additionnelles concernant P dans le bloc bi, et mi1, mi2, ..., miℓ les modes de procédures avec paramètres spécifiés dans ces déclarations.
- Soit P = T l'association et m' le mode du résultat de T.
- Soit k un entier.

Le processus d'identification donne d'abord à k la valeur 1, puis fonctionne de la façon suivante :

Pas 1 - S'il existe une et une seule déclaration pkj $\in$ {pk1, pk2, ..., pkℓ} telle que m' = (var + proc)*mkj, alors c'est l'origine correspondant à cette déclaration qui est identifiée.

Pas 2 - S'il existe au moins deux déclarations pki, pkj $\in$ {pk1, pk2, ..., pkℓ} telles que m' = (var + proc)*mki et m' = (var + proc) mkj, alors le programme est erroné à cause d'une ambiguïté d'identification.

Pas 3 - S'il n'existe pas une telle déclaration, et si k = n, alors le programme est erroné car aucune identification n'est possible. Si k < n, on retourne au pas 1 avec k ayant la valeur k + 1.

Lorsque l'identification a été faite, on effectue (EX + AP)*(mkj, T) qui est nécessairement VRAI ici.

Le mode d'une association est le mode des valeurs qui peuvent être associées à l'origine qu'elle utilise.

Les associations multiples :

O1 = O2 = ... = On = T

sont comprises entre :

O1 = (O2 = (... (On = T) ...))

Le mode de l'ensemble est alors, évidemment, celui de l'association la plus extérieure. Ainsi, si W est un identificateur identifiant une origine à laquelle peuvent être associées des valeurs de mode var proc ent, P de mode proc ent et I de mode ent, on peut écrire :

I = P = W = <un terme T à résultat de mode m ⇒ var proc ent>

Ceci sera finalement transformé en :

I = ap(ent, (P = ex(proc ent, (W = f(var proc ent, T)))))

et le résultat de l'association est de mode ent.

G.2.2. Dynamique

Soit une association :

O = T

T est d'abord exécuté et donne un résultat V. V est alors associé à l'origine identifiée par O, et devient la valeur de l'association.

G.2.3. Commentaires

L'association permet de "faire varier" la valeur associée à une origine identifiée par un identificateur, ou à une partie d'une origine dans le cas de l'identification d'un champ de structure. On note que l'identification de l'origine intéressé par une association à une procédure générique tient compte du mode du résultat. Ceci permet par exemple d'écrire, sans problème d'ambiguïté :

```
(P est générique dans ...
    (P estaussi proc(ent) ent dans ...
        (P estaussi proc(ent) reel dans ...
            P = un terme à résultat de mode
                proc(ent) ent ; ...
            P = un terme à résultat de mode
                proc(ent) reel, ... ) ... ) ... )
```

D'autre part, on voit que la "variabilité" apportée par l'association est soumise à des limites strictes dues au fait qu'une procédure ne peut pas associer de valeur à ce qui est déclaré à son extérieur. Comme on le verra au paragraphe M.2.2.2. ceci est une conséquence de la volonté de supprimer en BASEL tous les problèmes de portée qui se posent en Algol 68. Cependant, on verra que les variables (objets de mode var m) apportent toute la liberté que l'on peut désirer sur ce problème de "variabilité", mais toujours sans aucun risque de difficulté quant à la question des portées.

H. Les termes

H.1. Syntaxe

terme : selection|
facteur
selection : <identificateur de> ... identificateur de facteur

EXEMPLES :

Dans la portée des déclarations :

S1 est struct(ent a, struct(ent i, bool j) b),
S2 est var struct(ent n, reel x),

On peut écrire :

a de S1 ; b de S1 ; i de b de S1 ;
n de S2.

H.2. Sémantique

H.2.1. Statique

Une sélection a la forme I de A, où I est un identificateur et A un facteur à résultat de mode m.

Les sélections multiples :

I1 de I2 de ... de Ip de A.

sont comprises comme :

$$I1 \underline{de} (I2 \underline{de} (\ldots \underline{de} (Ip \underline{de} A) \ldots))$$

Comme condition préalable, il faut que :

$$m = (\underline{var} + \underline{proc})^* m'$$

avec m' = struct(m1 s1, m2 s2, ..., mn sn), et qu'il existe un entier j ($1 \leq j \leq n$) tel que I = sj.
Cependant, la transformation $(ex + ap)^*$ n'est pas toujours appliquée "jusqu'au bout", car deux cas sont possibles.

1er cas : Si $m = (\underline{var} + \underline{proc})^* \underline{var}\ m'$, alors on effectue :

$$(EX + AP)^*(\underline{var}\ m', A)$$

qui doit être VRAI.
Il s'agit alors d'une sélection de variable, et le résultat de la sélection est de mode var mj.

2ème cas : Si $m = (\underline{var} + \underline{proc})^* \underline{proc}\ m'$,
ou si m = m', alors on effectue :

$$(EX + AP)^*(m', A)$$

qui doit être VRAI.
Il s'agit alors d'une sélection de champ, et le résultat de la sélection est de mode mj.

H.2.2. Dynamique

Soit m' le mode de la structure impliquée dans la sélection :

$$m' = \underline{struct}(m1\ s1, m2\ s2, \ldots, mn\ sn).$$

Soit sj l'identidicateur utilisé :

- Pour une sélection de variable, si la valeur de mode var m' est la valeur nulle de ce mode, alors le programme est erroné.
  Si la valeur de mode var m' n'est pas la valeur nulle de ce mode, le résultat est la variable allouée au champ portant le sélecteur sj. (Voir le paragraphe J.2.2.1. pour l'allocation des variables).
- Pour une sélection de champ, le résultat de la sélection est le champ portant le sélecteur sj.

I. Les facteurs

I.1. Syntaxe

facteur : itération |
primaire
itération : <nombre fois> ... nombre fois primaire
nombre : primaire

EXEMPLES :

Dans la portée des déclarations :

N est ent, FACTN est ent, I est ent, S est seq ent

On peut écrire :

```
FACTN = I = 1 ;
N fois (FACTN = prod (FACTN, I) ; I = add (I, 1) ; vide)
```

Ceci associe à FACTN la valeur de N!

On peut aussi écrire :

```
FACTN = I = 1 ;
S = N fois (FACTN = prod (FACTN, I), I = add (I, 1) ; vide)
```

Ceci associe à S une séquence de longueur N dont la première variable retient 1!, la seconde 2!, ..., la $N^{ème}$ N!.

I.2. Sémantique

I.2.1. Statique

Soit une itération de la forme :

N fois P

N et P étant des primaires de modes respectifs mn et mp.
On effectue d'abord :

(EX + AP)*(ent, N)

qui doit être VRAI pour que le programme soit correct.

Si P est à résultat vide, l'itération est à résultat vide. Il s'agit alors d'une boucle.
Si P n'est pas à résultat vide, l'itération est à résultat de mode seq mp. Il s'agit alors d'une création de séquence.

Les itérations multiples :

N1 fois N2 fois ... Nn fois P

sont comprises comme :

N1 fois (N2 fois (... fois (Nn fois P) ...))

I.2.2. Dynamique

Soit N le "nombre" d'une itération et P son primaire, à résultat de mode mp.

L'exécution d'une itération commence par l'exécution de son "nombre" qui a comme résultat un entier ℓ. (Ce ne doit pas être la "valeur nulle" de mode ent).

- Cas d'une boucle

  - Si $\ell \leq 0$, l'exécution de l'itération est terminée.
  - Si $\ell > 0$, tout se passe comme si, à la place de l'itération, on avait écrit une expression composée sérielle de la forme :

$$\underbrace{(P, P, \ldots, P)}_{\ell \text{ fois}}$$

- Cas d'une création de séquences

  - Si $\ell \leq 0$, le résultat est une séquence nouvellement créée, de mode seq mp et de longueur nulle, c'est-à-dire ne contenant aucune variable.
  - Si $\ell \geq 1$, tout se passe comme si, à la place de l'itération, on avait écrit une modification explicite de la forme :

seq mp [P] si $\ell = 1$
seq mp$\underbrace{(P, P, \ldots, P)}_{\ell \text{ fois}}$ si $\ell > 1$

Ceci (voir le paragraphe L.2.2.6.) crée une séquence de longueur ℓ, dont la première variable retient le résultat de la première exécution de P, la deuxième variable le résultat de la deuxième exécution de P, etc....

I.2.3. Commentaires

BASEL n'a pas les "boucles pour" évoluées des langages de la famille Algol. Encore une fois, il ne faut pas oublier qu'il s'agit avant tout d'un langage de base. Cependant, l'itération de BASEL permet, d'une part, de reconstruire la plupart des mécanismes d'itération d'autres langages. Ainsi, dans l'exemple du calcul de N! donné plus haut on est parvenu à construire très simplement une boucle tout à fait classique. On note que le primaire écrit après fois dans cette boucle a la forme d'une expression composée à un seul bloc à résultat vide, afin d'éviter de provoquer inutilement la création d'une séquence. D'autre part, l'itération de BASEL peut avoir un résultat sous la forme d'une séquence. Ceci offre des possibilités que peu de langages peuvent proposer. Un exemple en est donné plus haut. Un autre exemple est la construction d'une matrice triangulaire supérieure T, de rang N, dont les éléments non nuls de la ligne i sont égaux à i :

```
(T est seq seq ent, N est ent dans ...
 T = (I est ent, J est ent dans
                  I = 0 ; J = add (N, 1) ;
                  N fois (I = add (I, 1) ;
                            (J = sous (J, 1)) fois I)) ...)
```

J. Les primaires

J.1. Syntaxe

```
primaire : operation|
           operande
operation : <alloc | affect | meme|
            change | long|
            ou | et | non|
            eg | neg | inf | infeg | supeg | sup|
            add | sous | prod | div | quot | rest | puiss|
            abs | sign | pair | moins|
            arr | pent | flott|
            entlu | reellu | boollu | carlu | chainelue | taper|
            imprimer > operande
```

EXEMPLES :

Dans la portée des déclarations :

```
I est ent, X est var reel, S est seq ent
```

on peut écrire :

```
I = entlu ("I = ") ;
X = alloc 3.14159 ;
affect (X, 2.71828) ;
S = [3, 4, 5] ;
S = change (S, 2) ;
affect (S(4), 6) ;
(et (eg (long S, 5), eg (I, arr X)) → I = add (I, long S)) ;
imprimer (infeg(I, 0) → "NEGATIF ou NUL"|" POSITIF")
```

J.2. Sémantique

J.2.1. Statique

On distingue 7 sortes d'opérations :

- des opérations sur les variables
- des opérations sur les séquences
- des opérations sur les booléens (opérations logiques)
- des opérations de comparaison
- des opérations arithmétiques
- des opérations de conversion
- des opérations (rudimentaires) de lecture-écriture.

La forme générale d'une opération est :

op O

où op est "l'opérateur" qui indique de quelle opération il s'agit, et O un opérande d'un certain mode noté m dans ce qui suit. O ne doit jamais être à résultat vide.

J.2.1.1. Opérations sur les variables

- L'allocation -

L'allocation est l'opération de création d'une variable :

alloc O

Elle a un résultat de mode var m.

- L'affectation -

L'affectation est l'opération qui fait retenir une valeur par une variable. Soit un mode m' quelconque, un mode m1 = proc* var m' et un mode m2 quelconque. On vérifie alors que m = (var + proc)* tuple(m1, m2), puis on effectue :

TT(tuple (var m', m'), (ex + ap)* (tuple(m1, m2), O)),

ce qui doit être VRAI pour que le programme soit correct.
Une affectation a un résultat de mode var m'.

- L'identité -

L'identité sert à comparer deux valeurs d'un même mode var m'.

Soient deux modes m1 et m2. Le mode m' est défini comme suit :

- m1 et m2, qui ne sont pas nécessairement égaux, peuvent tous les deux être mis sous la forme :

$$(\underline{var} + \underline{proc})^* var\ m'$$

- Il n'existe pas d'entier p tel que m1 et m2 puissent également tous les deux être mis sous la forme :

$$(\underline{var} + \underline{proc})^*\ \underline{var}^p\ \underline{var}\ m'$$

- Il n'existe pas d'entier q tel que m1 et m2 puissent également tous les deux être mis sous la forme :

$$(\underline{var} + \underline{proc})^*\ \underline{var}\ \underline{proc}^q\ \underline{var}\ m'$$

On vérifie alors que m = (var + proc)* tuple(m1, m2), puis on effectue :

TT(tuple (var m', var m'), (ex + ap)* (tuple(m1, m2), 0)).

ce qui doit être VRAI pour que le programme soit correct.
Une identité a un résultat de mode bool.

J.2.1.2. Opérations sur les séquences

- Le changement -

Le changement sert à rajouter ou à supprimer un certain nombre de variables dans une séquence.

Soient m' un mode quelconque et m1 un mode de la forme (var + proc)* seq m'. On vérifie alors que m est de la forme (var + proc)* tuple(m1, m2), puis on effectue :

TT(tuple(seq m', ent), (ex + ap)* (tuple(m1, m2), 0)),

ce qui doit être VRAI pour que le programme soit correct.
Un changement a un résultat de mode seq m'.

- La longueur -

L'opération long 0 a pour résultat la longueur d'une séquence obtenue en résultat de son opérande sur lequel on a effectué :

$$(EX + AP)^* \text{ (seq } m', 0)$$

ce qui doit être VRAI, m' étant un mode quelconque.
Le résultat de cette opération est de mode ent.

J.2.1.3. Opérations logiques

Les opérations logiques ont toutes un résultat de mode bool.
Pour la disjonction (ou) et la conjonction (et) les modifications applicables sont définies par :

$$F \text{ (tuple (bool, bool), 0)}$$

qui doit être VRAI.
Pour la négation (non), les modifications sont définies par :

$$(EX + AP)^* \text{ (bool, 0)}$$

qui doit être VRAI.

J.2.1.4. Opérations de comparaison

Les opérations de comparaison ont toutes un résultat de mode bool.
Pour l'égalité (eg) et non-égalité (neg) il faut que l'une des quatre fonctions suivantes soit VRAIe :

$$F \text{ (tuple (bool, bool), 0)}$$
$$\text{ou } F \text{ (tuple (ent, ent), 0)}$$
$$\text{ou } F \text{ (tuple (reel, reel), 0)}$$
$$\text{ou } F \text{ (tuple (car, car), 0)}$$

Ces fonctions étant VRAIes de façon exclusive, ce sont évidemment les transformations correspondant à celle qui est VRAIe qui sont appliquées.

De même, l'une des deux fonctions suivantes doit être VRAIe pour les comparaisons :

- inférieur (inf)
- inférieur ou égal (infeg)

- supérieur ou égal (supeg)
- supérieur (sup) :

Ce sont :

F (tuple (ent, ent), 0)
ou F (tuple (reel, reel), 0)

J.2.1.5. Opérations arithmétiques

On distingue les opérations arithmétiques à deux arguments et les opérations arithmétiques à un argument.

- Opérations arithmétiques à deux arguements -

Ce sont : - l'addition (add)
- la soustraction (sous)
- le produit (prod)
- la division réelle (div)
- le quotient entier (quot)
- le reste de la division entière (rest)
- l'élévation à une puissance entière (puiss)

Pour add, sous et prod si :

F (tuple (ent, ent), 0)

est VRAI, le résultat est de mode ent.

Si : F (tuple (reel, reel), 0)

est VRAI, le résultat est de mode reel.

Pour div il faut que :

soit F (tuple (ent, ent), 0)
soit F (tuple (reel, reel), 0)

soit VRAI. Dans les deux cas, le résultat est de mode reel.

Pour quot et pour rest, il faut que :

F (tuple, (ent, ent), 0)

soit VRAI. Le résultat est de mode ent.

Pour puiss, il faut que :

F (tuple (ent, ent), 0)
ou F (tuple (reel, ent), 0)

soit VRAI. Dans le premier cas, le résultat est de mode ent, dans le second il est de mode reel.

- Opérations arithmétiques à un argument -

Ce sont : - la valeur absolue (abs)
- le signe (sign)
- la parité (pair)
- le changement de signe (moins).

Pour abs et moins, si

$(EX + AP)^*$ (ent, 0)

est VRAI, le résultat est de mode ent.
Si :

$(EX + AP)^*$ (reel, 0)

est VRAI le résultat est de mode reel.

Pour sign, il faut que :

$(EX + AP)^*$ (ent, 0)
ou $(EX + AP)^*$ (reel, 0)

soit VRAI, et le résultat est toujours de mode ent.

Pour pair, il faut que :

$(EX + AP)^*$ (ent, 0)

soit VRAI, et le résultat est de mode bool.

J.2.1.6. Opérations de conversion

Pour l'arrondi (arr) et la partie entière (pent) il faut que :

$(EX + AP)^*$ (reel, 0)

soit VRAI, et le résultat est de mode ent.

Pour la conversion d'entier à réel (flott), il faut que :

$$(EX + AP)^* \text{ (ent, 0)}$$

soit VRAI, et le résultat est de mode reel.

J.2.1.7. Opérations de lecture-écriture

Le problème des entrées-sorties n'a pas du tout été abordé en profondeur en BASEL, d'autres composants de ELF devant éventuellement se charger de cette activité et de toutes les questions relatives à la manipulation de fichiers. Cependant, avant la disponibilité d'un système général, il a été jugé utile de prévoir quelques opérations très rudimentaires qui permettent la lecture sur un terminal-machine à écrire, et l'écriture soit sur ce même terminal, soit sur une imprimante rapide, le terminal et l'imprimante étant tous les deux choisis de façon standard. D'autre part, tous les formats de lecture et d'écriture sont eux aussi standards.

Il y a cinq opérations de lecture, pour lesquelles, il faut que :

$$F \text{ (seq car, 0)}$$

soit VRAI.

Le résultat d'entlu, lecture d'un entier, est de mode ent.
Le résultat de réellu, lecture d'un reel, est de mode reel.
Le résultat de boollu, lecture d'un booléen, est de mode bool.
Le résultat de carlu, lecture d'un caractère, est de mode car.
Le résultat de chainelue, lecture d'une chaîne (voir paragraphe N) est de mode seq car.

Il y a deux opérations d'écriture, taper, pour écrire sur le terminal, et imprimer, pour écrire sur l'imprimante. Pour chacune d'elle il faut que :

- soit $(EX + AP)^*$ (ent, 0) soit VRAI, auquel cas le résultat est de mode ent : c'est une écriture d'entier,
- soit $(EX + AP)^*$ (reel, 0) soit VRAI, auquel cas le résultat est de mode reel : c'est une écriture de réel.
- soit $(EX + AP)^*$ (bool, 0) soit VRAI, auquel cas le résultat est de mode bool : c'est une écriture de booléen.

- soit (EX + AP)* (car, 0) soit VRAI, auquel cas le résultat est de mode car : c'est une écriture de caractère.
- soit F (seq car, 0) soit VRAI, auquel cas le résultat est de mode seq car : c'est une écriture de chaîne.

J.2.2. Dynamique

L'exécution de toutes les opérations commence par l'exécution de leur opérande, qui donne un résultat V de mode m.

J.2.2.1. Opérations sur les variables

- L'allocation -

Une nouvelle variable, de mode var m, et créée, différente de toutes celles qui ont été créées jusqu'alors, et la valeur V est retenue par cette variable.
C'est cette variable qui est le résultat de l'allocation.
De plus, lorsque la variable créée est de mode :

var struct (m1 s1, m2 s2, ..., mn sn)

une variable est allouée à chaque champ, c'est-à-dire que, en plus de la variable obtenue en résultat de l'allocation, n autres variables sont créées :

- une variable de mode var m1, qui retient le champ s1 de V ;
- une variable de mode var m2, qui retient le champ s2 de V ;

...

- une variable de mode var mn, qui retient le champ sn de V.

- L'affectation -

Si la variable qui constitue le premier élément du tuple V est une valeur nulle, le programme est erroné. Si tel n'est pas le cas, la valeur qui constitue le deuxième élément du tuple V est retenue par la variable qui en constitue le premier élément.

De plus, dans le cas où la variable est de mode :

var struct(m1 s1, m2 s2, ..., mn sn)

la valeur de chaque champ de la valeur du deuxième élément du tuple V est retenue par la variable allouée au champ correspondant de la variable.
C'est la variable, premier élément du tuple V qui est le résultat de l'affectation.

- L'identité

Les deux éléments du tuple V sont comparés. S'il s'agit des mêmes variables le résultat est vrai, sinon il est faux.

J.2.2.2. Opérations sur les séquences

- Le changement

Soit s la séquence de mode seq m' qui est le premier élément du tuple V et n l'entier qui en est le deuxième élément.
Si au moins l'un de ces deux éléments est la valeur nulle de son mode, le programme est erroné.
Soient ℓ la longueur de s et v1, v2, ..., vℓ les variables qui, dans cet ordre, composent s.

Si n > 0, n nouvelles variables de mode var m' sont créées, différentes de toutes celles créées jusqu'alors. Soient vℓ+1, vℓ+2, ..., vℓ+n ces variables.
Une nouvelle séquence s' est alors créée, de longueur ℓ+n, et dont les variables sont dans l'ordre suivant :

v1, v2, ..., vℓ, vℓ+1, ..., vℓ+n.

La séquence s' est le résultat du changement.

Si n = 0, la séquence s est le résultat du changement.

Si n < 0, et si ℓ-|n|≤0, une nouvelle séquence de mode seq m' et de longueur nulle est créée. Cette séquence est le résultat du changement.

Si n < 0, et si ℓ-|n|>0, une nouvelle séquence, de longueur ℓ-|n| est créée, dont les variables sont :

v1, v2, ..., vℓ-|n|.

Cette séquence est le résultat du changement.

- La longueur -

Si V est la valeur nulle de mode seq mi, le programme est erroné.
Sinon, le résultat est longueur de la séquence V.

J.2.2.3. Opérations logiques

Soient a et b les éléments du tuple V, les résultats obtenus sont :

- la conjonction de a et b pour et.
- la disjonction de a et b pour ou.

Pour non, le résultat est la négation de V.
Si soit a, soit b, soit les deux sont une valeur nulle, le programme est erroné.

J.2.2.4. Opérations de comparaison

Soient a et b les éléments du tuple V.

- pour eg, le résultat est vrai si a est égal à b, sinon faux.
- pour neg, le résultat est vrai si a n'est pas égal à b, sinon faux.
- pour inf, le résultat est vrai si a est inférieur à b, sinon faux.
- pour infeg, le résultat est vrai si a est inférieur ou égal à b, sinon faux.
- pour supeg, le résultat est vrai si a est supérieur ou égal à b, sinon faux.
- pour sup, le résultat est vrai si a est supérieur à b, sinon faux.

Si soit a, soit b, soit les deux sont une valeur nulle, le programme est erroné.

J.2.2.5. Opérations arithmétiques

Soient a et b les éléments du tuple V pour les opérations à deux arguments.
Les résultats sont :

- pour add : la somme de a et b.
- pour sous : la différence a-b.
- pour prod : le produit de a par b.
- pour div : la division réelle de a par b.
- pour quot : le quotient entier de a par b.

- pour rest : le reste du quotient entier de a par b.
- pour puiss : l'élévation de a à la puissance b.
- pour abs : la valeur absolue de V.
- pour sign : -1 si V est négatif, 0 s'il est nul et 1 s'il est positif.
- pour pair : vrai si V est pair, faux autrement.
- pour moins : l'opposé de V.

Dans tous les cas, si au moins l'un des arguments est une valeur nulle, le programme est erroné.

J.2.2.6. Opérations de conversion

- pent a pour résultat la partie entière de V.
- arr a pour résultat la partie entière de add(V, 0.5)
- flott a pour résultat le reel "équivalent" à V, le passage du mode ent au mode reel n'étant qu'un changement de représentation interne : de virgule fixe à virgule flottante par exemple.

Si V est une valeur nulle, le programme est erroné.

J.2.2.7. Opérations de lecture-écriture

- Opérations de lecture -

Les opérations de lecture effectuent d'abord un retour chariot sur le terminal puis elles écrivent sur la ligne ainsi commencée un point d'interrogation (?) suivi de la suite des caractères retenus par les variables successives de la séquence V. Ceci indique à l'utilisateur qu'une lecture est attendue, et éventuellement de quelle lecture il s'agit. Le clavier est alors débloqué et l'utilisateur doit taper une notation de valeur, suivi d'un retour chariot à la suite duquel le clavier est à nouveau bloqué :

- pour entier, il doit taper une notation d'entier. L'entier correspondant devient le résultat de l'opération.
- pour reellu, il doit taper une notation de réel. Le réel correspondant devient le résultat de l'opération.

- pour boolu, il doit taper une notation de booléen. Le booléen correspondant devient le résultat de l'opération.
- pour carlu, il doit taper une marque (voir le paragraphe N). Le caractère correspondant devient le résultat de l'opération.
- pour chainelue, il doit taper un corps de chaîne (voir le paragraphe N). La valeur de mode seq car obtenue à partir de la chaîne ayant ce corps est le résultat de l'opération.

Un simple retour chariot est une erreur dans les cas de entlu, reellu, boollu et carlu et il crée une chaîne vide dans le cas de chainelue.

- Opérations d'écriture -

Si l'opération est taper, l'écriture se fera sur le terminal. Si l'opération est imprimer, l'écriture se fera sur l'imprimante rapide.

Dans les deux cas, on considère que l'écriture se fait dans un livre, qui a des pages elles-mêmes constituées de lignes contenant des cases. Il y a donc deux livres, notés conventionnellement LT pour le terminal et LI pour l'imprimante. Une case de LT est alors repérée par LT(pt, ℓt, ct) et une case de LI par LI(pi, ℓi, ci), où pt, pi, ℓt, ℓi, ct, ci sont des entiers positifs utilisés ici pour représenter conventionnellement :

- le numéro de page courante du terminal pour pt ;
- le numéro de page courante de l'imprimante pour pi ;
- le numéro de ligne courante du terminal pour ℓt ;
- le numéro de ligne courante de l'imprimante pour ℓi ;
- le numéro de case courante du terminal pour ct ;
- le numéro de case courante de l'imprimante pour ci.

Au début de l'exécution pt = pi = ℓt = ℓi = ct = ci = 1.

Par la suite, pt et pi croissent uniformément, ℓt croît modulo maxlt, qui est le nombre de lignes dans une page de LT, ℓi croît modulo maxli, qui est le nombre de lignes dans une page de LI, ct et ci peuvent croître et décroître, à condition que l'on ait toujours : $1 \leq ct \leq maxct$ et $1 \leq ci \leq maxci$ où maxct et maxci sont respectivement le nombre de cases dans une ligne de LT et le nombre de cases dans une ligne de LI.

Dans ce qui suit, on appellera L le livre (LT ou LI) sur lequel va s'effectuer l'écriture, p le numéro de page courante de ce livre (pt ou pi), ℓ le numéro de ligne (ℓt ou ℓi), c le numéro de case (ct ou ci), maxl le nombre de lignes dans une page (maxlt ou maxli) et maxc le nombre de cases dans une ligne (maxct ou maxci).

On appellera S :

- s'il s'agit d'une écriture d'entier, la suite de marques qui constitue la notation de l'entier à écrire.
- s'il s'agit d'une écriture de réel, la suite de marques qui constitue la notation du réel à écrire.
- s'il s'agit d'une écriture de booléen, la suite de marques qui constitue la notation du booléen à écrire.
- s'il s'agit d'une écriture de caractère, la marque correspondant à ce caractère.
- s'il s'agit d'une écriture de chaîne, la suite des marques correspondant aux caractères retenus par les variables de la séquence V.

Soit n le nombre de marques qui constituent S.

Si c+n-1 < maxc, les n marques de S sont écrites successivement dans les cases c, c+1, ..., c+n-1 de la ligne ℓ de la page p, et c prend la valeur c+n.

Si c+n-1 = maxc, les n marques de S sont écrites successivement dans les cases c, c+1, ..., c+n-1 de la ligne ℓ de la page p, et il y a un saut à la ligne (voir plus bas).

Si c+n-1 > maxc, et si n < maxc, il y a d'abord un saut à la ligne, puis les n marques de S sont écrites successivement dans les cases 1, 2, ..., n de la ligne de la page p, et c prend la valeur n+1.

Si c+n-1 > maxc, et si n ≥ maxc, il y a d'abord un saut à la ligne. Soit alors q le quotient de n par maxc et r le reste : il y a q remplissages de lignes en prenant successivement les marques de S, suivis chacun d'un saut à la ligne. Si r > 0, les r marques restantes sont écrites dans les cases 1, 2, ... r de ligne ℓ de la page p.

Le résultat d'une opération d'écriture est la valeur V de son opérande. Si cette valeur est une valeur nulle, la suite des marques de S est : •NUL•.
D'autre part, il ya trois cas spéciaux d'écritures qui servent à contrôler la mise en page. Pour chacun de ces cas, V est de mode seq car et devient la valeur -en général supprimée par le contexte- de l'opération.

Premier cas : positionnement dans la ligne. Pour ce cas particulier, V est de longueur 6 et les caractères retenus par ses variables successives correspondent à la suite de marque suivante :

•C•x y z

où x, y et z sont des chiffres formant la notation d'un entier n (commençant éventuellement par des zéros. Exemple •C•003). Il faut que :

$1 \leq n \leq maxc$

c prend alors la valeur n.

Deuxième cas : saut à la ligne. Pour ce cas particulier, V est de longueur 5 et les caractères retenus par ses variables successives correspondent à la suite de marques suivante :

•L•x y

où x et y sont des chiffres formant la notation d'un entier n. Il y a alors n sauts à la ligne successifs, un saut à la ligne se passant de la façon suivante :

- c prend la valeur 1.
- si $\ell \leq maxl$, ℓ prend la valeur $\ell+1$.
- si $\ell = maxl$, il y a un saut à page (voir ci-après).

Troisième cas : saut à la page. Pour ce cas particulier, V est de longueur 3, et les caractères retenus par ses variables successives correspondent à la suite de marques suivante :

•P•

Un saut à la page se passe de la façon suivante :

- c prend la valeur 1.
- ℓ prend la valeur 1.
- p prend la valeur p+1.

Enfin, dans le cas de l'écriture sur le terminal, il faut noter les interférences possibles avec les opérations de lecture. En effet, chaque opération de lecture commence par un saut à la ligne et se termine par un saut à la ligne, si bien que la mise en page de l'écriture est affectée par chaque lecture de la même façon que par deux sauts à la ligne commandés par :

•L•02

J.2.3. Commentaires

J.2.3.1. Opérations sur les variables

L'allocation remplit en BASEL un rôle analogue à celui des générateurs globaux d'Algol 68. Comme Algol 68, BASEL fait usage d'un "tas", c'est-à-dire d'une zone de mémoire gérée par des mécanismes de récupérations d'emplacements inaccessibles, et c'est dans ce "tas" que l'opération d'allocation réserve les emplacements pour ranger les valeurs retenues par les variables, les variables elles-mêmes pouvant être vues comme les "adresses" de ces emplacements. Dans ce "tas", et grâce aux mécanismes d'allocation et d'affectation, on peut construire des configurations de pointeurs aussi complexes qu'on le désire en utilisant des objets de mode var var m : c'est ce qui permet par exemple de construire des listes binaires du genre de celles de LISP ou des arbres à nombre de branches quelconque à chaque noeud.

Mais, en BASEL, il n'est pas possible de construire l'équivalent des noms locaux d'Algol 68. Aucun objet de mode var m en BASEL n'a son existence limitée par la terminaison de l'exécution du bloc où il a été créé. Quant aux origines, elles ne sont pas des objets manipulables aussi librement que des variables : elles sont seulement utilisées en tant que telles dans l'opération d'association , et dans l'opération d'accès (voir le paragraphe M.2.1.1.) quand l'origine concernée n'est pas déclarée à l'extérieur d'un texte de procédure où cet accès est éventuellement écrit (voir aussi M.2.2.2.). Ceci supprime complètement les problèmes de portées sur lesquels ont buté la plupart des implémenteurs d'Algol 68.

J.2.3.2. Opérations sur les séquences

En Algol 68, la flexibilité des rangs n'est qu'un leurre. En effet, on ne peut modifier la "longueur" d'un rang (ses bornes) qu'en lui affectant globalement tous les éléments d'un nouveau rang d'une autre "longueur".

En BASEL, les séquences sont vraiment flexibles, car on peut écrire par exemple :

S = change(S, 5)

S étant par exemple déclaré comme identifiant une origine à laquelle peuvent être associées des valeurs de mode seq ent.
Cette opération conserve les variables précédemment contenues par la séquence, et rajoute 5 nouvelles variables à leur suite, la nouvelle séquence ainsi créée étant à son tour associée à l'origine identifiée par S.

D'autre part, il ne semble pas que cette "vraie" flexibilité soit plus "chère" que celle d'Algol 68, en raison surtout du fait que toute séquence a pour éléments des variables, ce qui évite les recopies. Cependant, les séquences "non flexibles" n'existent pas en BASEL.

J.2.3.3. Opérations de comparaison et arithmétiques

C'est le mécanisme de construction des modes en BASEL et la structure des modes qu'il permet d'obtenir qui ont dicté la nature des diverses modifications élémentaires qui sont possibles. Ainsi il est "normal" qu'à var corresponde l'extraction de valeur, à proc l'appel implicite, etc.... Mais le choix des modes primitifs (ent, reel, bool, car) a été tout à fait arbitraire (bien qu'il tienne compte des objets primitifs que sait manipuler une machine). N'importe quel autre choix aurait pu être bon sans remettre en cause la structure du langage. C'est pourquoi, dans le mécanisme général des modifications n'entre pas la conversion -pourtant bien classique- d'entier à réel. Ceci a pour conséquence que l'on est forcé de préciser toujours, éventuellement en utilisant des opérations de conversion, le mode commun des arguments d'une opération de comparaison ou arithmétiques : tous les deux entiers ou tous les deux réels.

K. Les opérandes

K.1. Syntaxe

```
operande : application|
           element
application : element element element ...
```

EXEMPLES :

Dans la portée des déclarations :

```
sin est proc (reel) reel,
  x est reel,
  m est seq seq reel,
  v est seq reel
```

On peut écrire :

```
sin(x) ;
x = m(3) (4) ;
v = m(2) ;
affect(m(5)(6), sin(v(3)))
```

K.2. Sémantique

K.2.1. Statique

Soit une application de la forme E1 E2. (Une application multiple de la forme E1 E2 E3 ... Eq est comprise comme (... ((E1 E2) E3) ... Eq) .)

- Si E1 est une procédure générique, alors il s'agit d'un appel de procédure générique. Soit alors n2 le mode du résultat de E2.
- Sinon, soient n1 le mode du résultat de E1 et n2 le mode du résultat de E2 :
  - Si n1 = (var + proc)* proc(m1, m2, ..., mp) m alors il s'agit d'un appel de procédure.

- Si n1 = (var + proc)* seq m alors il s'agit d'une indexation de séquence.

K.2.1.1. Cas de l'appel de procédure générique

Soit P la procédure générique : parmi les origines dépendant de P, il faut identifier celle dont la valeur associée sera la procédure appelée. Ceci est accompli comme suit :

- Soit b1 le bloc où est écrit l'appel, b2 le bloc qui l'englobe, b3 le bloc qui englobe b2, etc... jusqu'à bn qui est soit le bloc où est déclarée la procédure générique P, soit l'un des blocs de l'expression composée qui constitue le programme.
- Soient Pi1, Pi2, ..., Piℓ les déclarations additionnelles concernant P dans le bloc bi, et mijr le mode du $r^{ème}$ paramètre de la procédure déclarée par Pij :
  - . Si cette procédure a un seul paramètre, on définit :

    mij = mij1
  - . Si cette procédure a q paramètres (q>1), on définit :

    mij = tuple(mij1, mij2, ..., mijq)
- Soit k un entier.

Le processus d'identification donne d'abord à k la valeur 1, et fonctionne alors de la façon suivante :

Pas 1. S'il existe une et une seule déclaration Pkj ∈ {Pk1, Pk2, ..., Pkℓ} telle que :

F(mkj, E2)

est VRAI, c'est la procédure associée à l'origine créée par cette déclaration qui sera appelée.

Soit m le mode du résultat de cette procédure tel qu'il est spécifié dans Pkj :

- Si m = rien, l'appel est à résultat vide.
- Sinon, l'appel a un résultat de mode m.

Pas 2. S'il existe au moins deux déclarations
Pkh, Pkj $\in$ {Pk1, Pk2, ..., Pkℓ}, telles que :

$$F(mkh, E2) \wedge F(mkj, E2)$$

est VRAI, alors le programme est erroné à cause d'une ambiguïté d'identification dans un appel.

Pas 3. S'il n'existe pas une telle déclaration et si k = n, alors le programme est erroné car aucune procédure ne répond à l'appel. Si k < n, on retourne au pas 1 avec k ayant la valeur k+1.

K.2.1.2. Cas de l'appel de procédure

On effectue : $(EX + AP)^*$ (proc(m1, m2, ..., mp) m, E1)
qui doit être VRAI pour un choix de modes m1, m2, ..., mp et m étant soit un mode soit rien.

- Si p = 1, il faut que : F(m1, E2) soit VRAI.
- Si p > 1, il faut que : F(tuple(m1, m2, ..., mp), E2) soit VRAI.
- Si m = rien, l'appel est à résultat vide, sinon il a un résultat de mode m.

K.2.1.3. Cas de l'indexation de séquence

On effectue : $(EX + AP)^*$ (seq m, E1) qui doit être VRAI pour un certain mode m.

- Si n2 = $(var + proc)^*$ ent, on effectue : $(EX+AP)^*$ (ent, E2) qui est alors VRAI, et le résultat de l'indexation est de mode var m.
- Sinon, on effectue : F(tuple(ent, ent), E2) qui doit être VRAI, et le résultat de l'indexation est de mode seq m.

K.2.2. Dynamique

Soient V1 et V2 les résultats respectifs de E1 et E2, exécutés collatéralement.

K.2.2.1. Cas de l'appel de procédure générique

V1 est la procédure associée à l'origine qui a été identifiée parmi les origines rattachées à la procédure générique. Tout se passe comme pour l'appel de procédure.

K.2.2.2. Cas de l'appel de procédure

- Si V1 est une valeur nulle, le programme est erroné.
- Sinon :
  - . Soit D la liste des déclarations des paramètres de la procédure V1. D a la forme :

    d1 est m1, d2 est m2, ..., dp est mp

    d1, d2, ..., dp étant des identificateurs.
  - . Soit I la partie instruction de la procédure V1.

Alors :

- Si p = 1, tout se passe comme si, à la place de l'appel, on avait écrit :

  (D dans d1 = V2 ; I)
- Si p > 1, soient W1, W2, ..., Wp les éléments, dans cet ordre, du tuple V2. Tout se passe alors comme si à la place de l'appel on avait écrit :

  (D dans d1 = W1 ; d2 = W2 ; ... ; dp = Wp ; I)

K.2.2.3. Cas de l'indexation de séquence

- Si soit V1, soit V2, soit les deux sont des valeurs nulles, le programme est erroné.
- Sinon, soit ℓ la longueur de la séquence V1.
  - . Si V2 est un entier, il faut que $1 \leq V2 \leq \ell$, sinon le programme est erroné. Le résultat de l'indexation est alors la variable de rang V2 dans la séquence V1.
  - . Si V2 est un tuple à deux éléments entiers, W1 et W2, il faut que $1 \leq W1 \leq \ell$ et $1 \leq W2 \leq \ell$, sinon le programme est erroné.

Si W2-W1 ≥ 0, le résultat est une séquence de longueur W2-W1+1 dont les variables sont les variables de rang. W1, W1+1, ..., W2 de V1.
Si W2-W1 < 0, le résultat est une séquence de longueur nulle, de même mode que V1.

K.2.3. Commentaires

Les procédures génériques permettent de déclarer et d'utiliser plusieurs procédures sous un même nom. Ceci permet par exemple d'appeler PLUS aussi bien l'addition d'entiers, de réels, que de complexes ou de matrices, l'identification, dans les cas usuels comme ceux-ci, ne posant pas de problème d'ambiguïté, car la conversion automatique d'entier à réel n'est pas une modification élémentaire en BASEL :

```
(PLUS est générique,
      PLUS estaussi proc(ent, ent) ent,
      PLUS estaussi proc(reel, reel) reel,
      PLUS estaussi proc(ent, reel) reel,
      PLUS estaussi proc(reel, ent) reel,
      complexe rep struct(reel PR, reel PI),
      PLUS estaussi proc(complexe, complexe) complexe,
      PLUS estaussi proc(complexe, reel) complexe,
                        etc...                        )
```

On doit également remarquer que l'association et l'appel sont les deux seuls endroits, avec les déclarations additionnelles, où il est possible d'écrire une procédure générique.

L'appel de procédure avec paramètres utilise un mécanisme uniforme de passage de paramètres qui permet de reconstruire ce qui est connu ailleurs sous les noms de passage "par valeur", passage "par référence", ou passage "par nom pur".

L'exemple suivant suffit pour résumer ces possibilités :

```
(p est proc(ent, var ent, proc ent) var ent, a est ent,
    b est var ent, c est proc ent dans
        a = 1 ;
        b = alloc 2 ;
        c = <... 3> ;
        p = <i est ent, j est var ent, k est proc ent dans
             affect(j, add(i, add(j, k)))> ;
        ... p(a, b, c) ; ...)
```

(les crochets < et > délimitent les textes de procédures ; voir le paragraphe M)

Dans l'appel p(a, b, c), a est passé "par valeur", b "par référence", et c "par nom pur". Après cet appel, la variable associée à l'origine identifiée par b retient la valeur 6.

D'autre part, si aux déclarations précédentes on avait rajouté :

```
t est tuple(ent, var ent, proc ent)
```

on pourrait alors choisir et former à l'avance la liste des paramètres d'un appel de p :

```
...
t = (a, b, c) ;
...
t = (3, alloc 5, c) ;
...
t = (c, p(a, b, c,), <...>)
...
```

et écrire l'appel :

```
p(t)
```

On peut choisir aussi, grâce à l'association, la procédure qui sera appelée.

Il faut remarquer enfin que l'application (appel ou indexation), est un endroit de la syntaxe de BASEL où, très naturellement, on se trouve amené à considérer le blanc comme significatif. Si bien que la règle est la suivante, similaire à celle qui régit les conventions de l'écriture courante :

- Deux identificateurs, un identificateur et une notation d'entier ou deux notations d'entier sont toujours séparés soit par au moins un blanc, soit par un symbole autre qu'une lettre ou un chiffre (qui, lui, peut être entouré de zéro ou plusieurs blancs, comme on veut).

Ainsi, dans l'exemple précédent, on aurait pu écrire l'appel final sous la forme :

```
p t
```

De même si on a déclaré :

```
sin est proc(reel) reel,
  x est reel
```

On peut écrire : `sin(x)`

mais aussi : `sin x`

Et encore, si on a déclaré :

```
s est seq seq ent
```

On peut écrire :

```
s(3)
s(3) (4)
s 3
s 3 4
s 3 (4)
...
```

Pour terminer, on peut noter que la syntaxe d'appel des procédures, combinée avec le mécanisme des modifications, et en particulier avec la mise en séquence, permet d'avoir en BASEL ce que l'on appelle quelquefois des procédures "à nombre variable de paramètres". En fait, il s'agit toujours d'un seul paramètre, qui doit être une séquence, ou une valeur modifiable en une séquence : le paramètre de l'appel peut donc être mis sous la forme d'une expression composée -sérielle ou collatéralle- dont le résultat est un tuple avec un nombre quelconque d'éléments.

Exemple :

```
(SOMME est proc(seq ent) ent dans
     SOMME = <V est seq ent dans
          (I est ent, J est ent dans
               I = 0 ; J = 0 ;
               long V fois(I = add(I, V(J = add(J, 1))) ; vide) ;
               I)> ;
     ...
     (S est seq ent, L est ent, M est ent, N est var ent,
      P est proc ent dans
          ...
          SOMME S ;
          ...
          SOMME(L, M, 3, sous (L, 2), N, P) ;
          ...
          SOMME(7, L, M) ;
          ...)            ...)
```

On voit donc qu'il n'y a aucun besoin ici de l'artifice des doubles parenthèses qui est nécessaire en Algol 68 : BASEL n'a pas de parenthèses spéciales pour les listes de paramètres d'un appel de procédure.

Sur le même sujet, l'exemple suivant reconstruit exactement le 'AND' de LISP : dès que l'un des paramètres est faux, on connait le résultat, qui est faux, et on arrête l'exécution à ce point, sans calculer la valeur booléenne des autres paramètres :

```
(ET est proc(seq proc bool) bool dans
     ET = <B est seq proc bool dans
          (I est ent dans I = 0 ;
          L : I = add(I, 1) ; P(I) → faux|
                              eg(I, long P) → vrai|
                                    allera L)> ;
...
(L est ent, M est ent, N est ent, B1 est bool, B2 est bool,
B3 est bool dans
     ...
     ET(<eg(L, M)>, <pair N>, <ou (B1, B2)>)
     ...
     ET(<B1>, <B2>, <non B3>,<sup(L, N)>,<ou (B3, non B1)>)
     ...)      ...)
```

L. Les éléments

L.1. Syntaxe

```
element : modification-explicite|
          base
modification-explicite : mode ... mode base
```

EXEMPLES :

Dans la portée des déclarations :

```
T est tuple(reel, reel),
complexe rep struct(reel PR, reel PI),
X est reel
```

On peut écrire :

```
complexe (3.14159, 2.71828)
complexe T
seq reel [3.14159, X, 2.71828, add (X, 0.8)]
```

L.2. Sémantique

L.2.1. Statique

Une modification explicite permet de provoquer le déclenchement d'une modification, ou d'une composition de modifications, en un endroit où le contexte seul ne suffirait pas à faire obtenir le résultat du mode voulu.

Soit une modification de la forme :

m B

où m est un mode et B la "base" utilisée.

On effectue :

F(m, B)

qui doit être VRAI, sinon le programme est erroné.
Le résultat est alors de mode m.

Les modifications multiples :

m1 m2 ...mn B

sont comprises comme :

m1 (m2 (... (mn B) ...))

L.2.2. Dynamique

Une modification explicite met en oeuvre les mécanismes mêmes qui sont utilisés par tout le système des modifications de BASEL. Il convient donc de décrire ici la sémantique dynamique des diverses modifications définies par la relation de compatibilité entre deux modes et de la suppression de valeur. Chacune de ces modifications fonctionne selon le principe suivant :

- elle part d'une valeur V, qui lui est fournie comme argument.
- elle produit un résultat R, ou bien elle a un résultat vide.

L.2.2.1. L'extraction de valeur

V est une variable. Si V est une valeur nulle, le programme est erroné.
Sinon, R est la valeur retenue par V.

L.2.2.2. L'appel implicite

V est une procédure sans paramètres. Si V est une valeur nulle, le programme est erroné.
Sinon, la partie instruction de V est exécutée, et R est son résultat, s'il existe.

L.2.2.3. La réunion

Deux cas, déterminés statiquement, sont possibles :

1ère cas. V n'est pas un objet de mode union. R est alors un objet de mode union qui prend la valeur V.

2ème cas. V est un objet de mode union. R est alors un objet de mode union, qui prend la valeur prise par V.

L.2.2.4. La modification de tuple

V est un tuple : soient v1, v2, ..., vn ses éléments. R est alors un tuple formé des éléments r1, r2, ..., rn, tels que ri ($1 \leq i \leq n$) est obtenu en faisant subir à vi une modification, ou une composition de modifications, déterminée statiquement. (Ce processus est récursif, ri pouvant par exemple être lui même un tuple).

L.2.2.5. La structuration de tuple

V est un tuple : soient v1, v2, ..., vn ses éléments. R est alors une structure, formée des champs r1, r2, ..., rn tels que ri ($1 \leq i \leq n$) est obtenu par une composition récursive de modifications appliquée à vi.

L.2.2.6. La mise en séquence

V est un tuple : soient v1, v2, ..., vn ses éléments. Chaque élément vi subit une modification, ou une composition de modification, déterminée statiquement, qui le transforme en une valeur v'i du mode des valeurs retenues par les variables de la séquence à construire.

Dans un premier temps, on obtient ainsi un tuple V' dont les éléments v'1, v'2, ..., v'n sont tous du même mode.

Dans un deuxième temps, il y a n allocations de variables, la $i^{ème}$ d'entre elles créant une variable qui retient la valeur v'i, et devenant la $i^{ème}$ variable de la séquence de longueur n ainsi construite.

L.2.2.7. La modification de structure

V est une structure. Soient v1, v2, ..., vn ses champs. R est alors une structure formée des champs r1, r2, ..., rn tels que ri ($1 \leq i \leq n$) est obtenu par une composition récursive de modifications appliquée à vi.

L.2.2.8. La suppression de valeur

V est une valeur quelconque, résultat d'une instruction I. Elle est "supprimée", c'est-à-dire que V, en tant que résultat de I, n'existe plus. Par exemple, dans :

```
... k = 1 ; ...
```

la valeur 1, en tant que résultat de k = 1, n'existe plus, bien qu'elle existe toujours en tant que valeur associée à l'origine identifiée par k.

L.2.3. Commentaires

Les modifications donnent à une valeur la "signification" qu'on lui attache "naturellement" dans le contexte où elle a été obtenue. Ainsi, si l'on écrit :

```
(X est var ent, Y est var ent dans
        ... affect (X, Y) ; ...)
```

aucune modification ne sera appliquée à X alors qu'une valeur sera extraite de Y. (Dans la terminologie de Strachey [Sr66] X sert à construire une

"L-value" et Y une "R-value". On peut d'ailleurs noter qu'en BASEL, les variables peuvent jouer aussi bien le rôle de "L-value" que celui de "R-value", tandis que les origines ne peuvent être que des "L-values").

De même, l'appel implicite de procédure a lieu quand on a besoin de lui pour donner son sens à une instruction. Par exemple :

```
(I est ent, P est proc ent, Q est proc ent dans
        ... I = P ; ... Q = P ; ...)
```

Ici, l'instruction I = P provoque un appel implicite de P pour en associer le résultat à I alors que Q = P, même lors de sa suppression de valeur, ne provoque aucun appel implicite ni de P ni de Q.

Cependant, si l'intervention automatique des modifications ne suffit pas à construire la valeur désirée, l'utilisation d'une modification explicite est nécessaire, ainsi que le montrent les quelques exemples suivants :

```
(complexe rep struct(reel PR, reel PI),
     T est tuple(reel, reel),
     X est reel,
     P est var var reel, v est union(seq reel, seq ent) dans
     ... P = alloc alloc 3.14159 ; ...
     ... affect (var reel P, 2.71828) ; ...
     ... X = PR de complexe T ; ...
     ... X = seq reel [3.14159, X, P] 2 ; ...
     ... U = seq ent (2, 3, 4) ; ...
```

M. Les bases

M.1. Syntaxe

```
base : accès|
       texte-de-procédure|
       notation|
       expression-composée
accès : identificateur
texte-de-procédure : < corps-de-procédure >
corps-de-procédure : [déclarations-de-parametres dans] partie-instructions
declarations-de-parametres : declaration-simple <, declaration simple> ...
```

(Les notations sont décrites plus loin, dans le paragraphe N).

EXEMPLE :

Dans la portée de :

```
concat est proc(seq car, seq car)
```

On peut écrire :

```
concat = <s1 est seq car, s2 est seq car dans
          (ℓ1 est ent, i est ent dans
           ℓ1 = long s1 ; i = 0 ; change(s1, long s2) ;
          long s2 fois
              (affect(s1 (add (ℓ1, add(i, 1))),
                      s2 (i = add(i, 1))) ; vide) ; s1)>
```

M.2. Sémantique

M.2.1. Statique

M.2.1.1. Accès

Lorsqu'une base est un accès, l'identificateur I utilisé doit avoir été déclaré dans un bloc ou un corps de procédure contenant, directement ou non, cet accès : soit b1 le bloc, ou corps de procédure, au niveau duquel est écrit l'accès, b2 le bloc, ou corps de procédure qui contient b1, ... etc ... jusqu'à bn qui est l'un des blocs de l'expression composée qui constitue le programme.

Soit un entier k, initialement égal à 1. La déclaration concernant I est trouvée de la façon suivante :

Pas 1 : Si une déclaration d'un identificateur I est parmi les déclarations du bloc bk, alors c'est cette déclaration qui est considérée.

Pas 2 : Sinon, si k = n, le programme est erroné à cause de l'utilisation d'un identificateur non déclaré. Si k < n, on revient au pas 1 avec k ayant la valeur k+1.

Si la déclaration considérée est soit une déclaration de générique, soit une déclaration additionnelle, I est une procédure générique et son identification est accomplie à l'aide des modes des paramètres de l'appel où elle est utilisée (voir le paragraphe K.2.1.1.) car c'est nécessairement dans un contexte d'appel qu'elle est écrite.

Sinon, l'identification est terminée est le résultat est du mode spécifié dans la déclaration.

M.2.1.2. Texte de procédure

Soit m le mode du résultat de la partie instruction du corps de procédure, s'il existe. Sinon m = rien.

Si le corps de procédure contient des déclarations de paramètres, soient m1, m2, ..., mn les modes spécifiés dans ces déclarations, dans cet ordre. Le mode de la procédure qui sera obtenue en résultat de l'évaluation du texte de procédure est alors proc(m1, m2, ..., mn) m.

S'il n'y a pas de déclarations de paramètres, ce mode est proc m.

M.2.2. Dynamique

M.2.2.1. Accès

Un accès identifie une origine : sa valeur est la valeur associée à cette origine.

M.2.2.2. Texte de procédure

La procédure, résultat d'un texte de procédure, est obtenue par évaluation de ce texte de procédure.

Une procédure contient éventuellement des déclarations de paramètres, et contient toujours une partie instructions.

Afin de décrire le processus d'évaluation d'un texte de procédure, il convient d'abord d'introduire une nouvelle possibilité syntaxique pour les "bases", possibilité qui ne peut être utilisée que dans la partie instruction d'une procédure, qu'il est impossible d'écrire dans un programme, et que seul le processus d'évaluation peut "écrire" : il s'agit de la valeur immédiate. Ainsi, à l'intérieur de la partie instruction d'une procédure (et non pas d'un texte de procédure) une base est définie syntaxiquement par :

base : accès|
texte-de-procédure|
notation|
expression-composée|
valeur-immédiate|

avec : valeur-immédiate : *v*

où v est effectivement une valeur, et non pas une entité syntaxique que l'on pourrait écrire dans un programme.

Dynamiquement, lors de l'appel d'une procédure, l'exécution de *v* a pour résultat v.

La notion de valeur immédiate étant ainsi définie, le processus d'évaluation d'un texte de procédure fonctionne de la façon suivante :

- Soit T le texte de procédure, qui a l'une des formes :

  <D dans I>
  ou <I>

  où D est une liste de déclaration et I une partie instruction.
- Soit {a1, a2, ..., an} l'ensemble des identificateurs utilisés dans I mais dont la déclaration se trouve à l'extérieur de T. (Comme on l'a vu au paragraphe G.2.1. aucun de ces identificateurs n'est utilisé comme origine simple dans une association : ce sont tous des accès).
- Soit vi la valeur associée à l'origine identifiée par ai ($1 \leq i \leq n$).
- Soit I' une copie de I.

Le processus d'évaluation de T construit alors une nouvelle partie instruction I" obtenue en remplaçant dans I' chaque occurrence de ai ($1 \leq i \leq n$) par la valeur immédiate *vi*. Il n'y a donc plus, dans I", d'identificateur dont la déclaration soit extérieure à T. Le résultat de l'évaluation de T est alors l'une des procédures :

<D dans I">

ou <I">

selon la forme de T.

M.2.3. Commentaires

L'évaluation des textes de procédures est l'une des clés de voutes de l'architecture de BASEL.

Tout d'abord, avec la règle interdisant l'association à des origines créées par des déclarations extérieures à une procédure, ceci permet de supprimer complètement les problèmes de portée. En utilisant une terminologie empruntée au rapport Algol 68, toutes les procédures de BASEL ont une portée globale, au même titre que les variables et les valeurs des modes primitifs. Ceci élargit l'utilisation des procédures à des cas interdits dans la plupart des langages, en particulier en Algol 68. Ainsi, on peut écrire une procédure qui construit la composition fonctionnelle de ses deux arguments, ceux-ci étant eux-mêmes des procédures, chose impossible en Algol 68.

Exemple :

```
(trigo rep proc(reel) reel,
     sin est trigo, cos est trigo,
     comp est proc(trigo, trigo) trigo,
          p est trigo, a est reel dans
          ...
     comp = <f est trigo, g est trigo dans
            <x est reel dans f(g(x))>> ; ...
          p = comp(sin, cos) ; ...
       a = comp (p, cos) (3.14159) ; ...)
```

On peut aussi obtenir, grâce à cela, certaines formes d'un mécanisme de protection. Par exemple, on peut construire une procédure qui effectue un ensemble quelconque d'actions représentées par un ensemble d'instruction I, et qui donne comme résultat le nombre de fois où elle a été appelée, le compteur utilisé étant complètement inaccessible de l'extérieur de la procédure (bien qu'il faille, nécessairement, le déclarer à son extérieur) :

```
(p est proc ent dans
     p = (c est var ent dans
               <I ; ent affect (c, add(c, 1))>) ; ...)
```

De plus, c'est grâce aux règles qui organisent la construction des procédures que la déclaration conditionnelle peut fonctionner en BASEL. En effet, s'il était autorisé, dans une procédure, d'associer des valeurs à des origines créées par des déclarations extérieures à cette procédure, on pourrait écrire :

```
(p est proc(ent) rien,
     u est union(bool, reel) dans

     ... p = <i est ent dans ... u = vrai ...> ;

... u = 3.14159 ; (u doitetre reel, x est reel dans
                        x = u ; ...  p(1) ; ...  x = u ; ...) ...)
```

Un tel programme, s'il était correct, conduirait à de véritables catastrophes, car la deuxième association x = u, bien que correcte statiquement, associerait un booléen à l'origine identifiée par x.

Enfin, il faut bien voir que c'est grâce aux variables que les procédures, malgré les règles strictes sur leur construction, conservent les possibilités qu'elles ont dans les autres langages : une procédure n'a pas le droit de changer la valeur associée à une origine créée à son extérieur, mais elle a accès à tout l'univers des variables.

Exemple :

```
(x est var reel, p est proc (ent) rien dans
                x = alloc 3.0 ;
          ... p = <i est ent dans ...
                              affect (x, 5.4) ; ... > ...)
```

C'est d'ailleurs par le biais des variables qu'il est possible de définir des procédures récursives. En effet, si l'on écrit :

```
(FACT est proc (ent) ent dans
    FACT = <I est ent dans
            eg (I, 0) → 1|
            prod (I, FACT (sous (I, 1)))>; ...)
```

il ne faut pas attendre de la procédure associée à l'origine identifiée par FACT le calcul de la factorielle de son paramètre. En effet, l'identificateur FACT utilisé dans le texte de procédure est remplacé, lors de l'évaluation de ce texte, par la valeur associée à l'origine qu'il identifie c'est-à-dire, ici, la valeur nulle de mode proc (ent) ent.

Par contre, si l'on écrit :

```
(FACTORIELLE est var proc(ent) ent dans
     FACTORIELLE = alloc nul proc(ent) ent ;
     affect (FACTORIELLE,
          <I est ent dans
           eg (I, 0) → 1|
           prod (I, FACTORIELLE (sous (I, 1)))>) , ...)
```

tout appel de la procédure retenue par la variable associée à l'origine identifiée par FACTORIELLE (appel qui peut s'écrire FACTORIELLE N grâce aux modifications) a pour résultat la factorielle de son paramètre. (Il est à noter que le corps de deux procédures de ces exemples - FACT et FACTORIELLE - ont exactement la même structure syntaxique).

N. Les notations

N.1. Syntaxe

notation : valeur-nulle|
vide|
notation-primitive|
chaîne

valeur-nulle : nul mode

vide : vide

notation-primitive : notation-d-entier|
notation-de-reel|
notation-de-booleen|
notation-de-caractere

notation-d-entier : chiffre ... chiffre

notation-de-reel : notation-d-entier . notation-d-entier

notation-de-booleen : vrai|
faux

notation-de-caractere : ' marque '

chaîne : " corps-de-chaîne "

corps-de-chaine : marque ...

marque : lettre|
chiffre|
autre

lettre : a b c — y z A B C — Y Z

chiffre : 0 1 2 — 8 9

autre : (|)|[|]|<|>|•|,|;|:|
+|-|*|/|=|≠|"|'|!|?|
¬||||_|%|@|&|$

EXEMPLES :

nul var ent
123 069
3.14159 0.07
'A' ',' '''' ''''
"" "A" "A!" """" "'"

N.2. Sémantique

N.2.1. Statique

N.2.1.1. Valeur nulle

Soit une valeur nulle de la forme :

nul m

son résultat est de mode n.

N.2.1.2. Vide

Comme son nom l'indique, vide a un résultat vide.

N.2.1.3. Notations primitives

Une notation d'entier a une valeur de mode ent.
Une notation de reel a une valeur de mode reel.
Une notation de booléen a une valeur de mode bool.
Une notation de caractère a une valeur de mode car. (pour un caractère, la marque correspondant à ' est '').

N.2.1.4. Chaîne

Une chaîne a une valeur de mode seq car. (Dans une chaîne, la marque correspondant à " est"").

N.2.2. Dynamique

N.2.2.1. Valeur nulle

Pour chaque mode, il existe une valeur distinguée unique appelée valeur nulle de ce mode.

Pour les modes primitifs, cette valeur est différente de toutes celles qui sont obtenues par les notations primitives.

D'autre part, m, m1, m2, ..., mn étant des modes, il existe une valeur nulle pour chacun des modes :

var m,
proc(m1, m2, ..., mn) m,
proc(m1, m2, ..., mn) rien,
proc m,
proc rien ,
seq m,
et union(m1, m2, ..., mn)

Pour les modes :

tuple(m1, m2, ..., mn)
et struct(m1 s1, m2 s2, ..., mn sn)

la valeur nulle est constituée des valeurs nulles des éléments ou des champs respectivement.

La notation :

nul m

construit la valeur nulle de mode m.

N.2.2.2. Vide

Rien .

N.2.2.3. Notations primitives

Les notations d'entiers, de réels, de booléens et de caractères ont leur signification habituelle.

N.2.2.4. Chaîne

Soit C1, C2, ..., Cn la suite des valeurs de mode car correspondant à la suite de marques qui constitue le corps de la chaîne.

- Si n = 0, le résultat est une nouvelle séquence, de mode seq car et de longueur nulle.
- Si n > 0, soit ki une notation de caractère pour le caractère Ci ($1 \leq i \leq n$).

Tout se passe alors comme si, à la place de la chaîne, on avait écrit :

seq car(k1, k2, ..., kn)

N.2.3. Commentaires

N.2.3.1. Valeur nulle

L'existence d'une notion de valeur nulle en BASEL permet de contrôler dynamiquement qu'une origine a bien eu une valeur associée à elle par une expression d'association. Mais cela permet aussi bien d'autres possibilités :

- une valeur nulle de mode var m est l'équivalent du nil d'Algol 68.
- une valeur nulle peut être utilisée chaque fois que l'on ne désire pas, ou que l'on ne peut pas préciser la valeur à utiliser, mais que l'on veut malgré tout fournir une valeur du mode correct. La valeur nulle a alors le rôle du fant d'Algol 68.

Exemple :

```
(p est var var ent dans
        p = alloc nul var ent ;
    ... affect(p, alloc nul ent) ; ...)
```

De même, il peut arriver qu'il soit plus simple dans certains cas, de ne pas préciser la valeur d'un ou plusieurs paramètres dans un appel de procédure.

N.2.3.2. Chaîne

La notation de chaîne a été introduite en BASEL pour deux raisons :

- il était souhaitable de pouvoir construire des suites de caractère de longueur nulle, pour pouvoir obtenir facilement toutes les opérations générales sur les chaînes de caractères.
- il est plus commode d'écrire : "ABC" que d'écrire
  seq car ('A', 'B', 'C'). C'est là une des rares concessions au "sucre syntaxique" qui ait été faite lors de la conception de BASEL.

3.1.3. INTRODUCTION A LA DEFINITION FORMELLE DE BASEL

Ainsi qu'il a été dit dans l'introduction à BASEL, la construction d'une définition formelle pour ce langage a sans cesse accompagné le travail de sa conception, dans le but de mettre à jour de façon aussi claire et complète que possible tous les effets éventuels de l'introduction de chaque concept dans le langage.

Ici, l'intention n'est pas de donner l'ensemble de cette définition car, en raison de la nature bi-dimensionnelle d'une partie du formalisme utilisé, son encombrement serait beaucoup trop grand pour cet ouvrage.

Cette définition est disponible sous une forme complète par ailleurs ([Ha69], [Jo69c] et [Jo69d]) pour une version américaine et non définitive de BASEL, qui ne contient pas tous les mécanismes décrits dans les paragraphes précédents. La version définitive - toujours américaine - de BASEL a également subi le contrôle de la définition formelle, mais celle-ci est restée sous la forme d'ajouts, de suppression ou de transformations de la définition antérieure, et il n'a pas été jugé utile de la mettre sous une forme publiable car elle avait rempli son rôle dans la forme où elle se trouvait.

Cependant, il demeure intéressant de décrire le formalisme et les techniques utilisées dans la construction de cette définition, non seulement à cause de leur relative originalité, mais aussi en raison de leur universalité qui permet d'envisager leur application à la plupart des langages de programmation, extensibles ou non.

3.1.3.1. Principe de la définition

La définition de BASEL est organisée selon le schéma, depuis lors devenu classique, d'un traducteur suivi d'un interpréteur. Le traducteur construit une représentation abstraite d'un programme, sous le contrôle d'une analyse syntaxique de la représentation concrète de ce programme. Lorsque le traducteur a terminé son travail, la représentation abstraite est confiée à l'interpréteur qui, sous le contrôle d'informations contenues dans cette représentation, effectue des actions qui transforment certaines parties de cette représentation elle-même. La syntaxe de BASEL et sa sémantique statique sont alors définies par le traducteur et sa sémantique dynamique par l'interpréteur.

Dans le traducteur, la syntaxe est décrite par un ensemble de règles de grammaire "context-free" et la sémantique statique par un ensemble de procédures qui ont également pour rôle de construire la représentation abstraite du programme, et dont l'appel est provoqué par la reconnaissance de certaines structures syntaxiques.

Dans l'interpréteur, la sémantique dynamique est entièrement décrite par un ensemble de procédures qui s'appellent mutuellement, et qui effectuent leurs actions respectives pendant un parcours de la représentation abstraite construite par le traducteur. L'appel initial de l'interpréteur est provoqué par la reconnaissance syntaxique d'un programme complet.

Le formalisme utilisé a donc plusieurs composantes :

- les règles de grammaire context-free ;
- l'appel des procédures de la sémantique statique, et son couplage avec les règles de grammaire ;
- la description des procédures de la sémantique statique ;
- la description des procédures de la sémantique dynamique.

Les trois premières composantes constituent le traducteur, et la dernière l'interpréteur. D'autre part, comme on le verra par la suite, c'est un même langage, appelé AMBIT/G, qui est utilisé pour décrire les procédures de la sémantique statique et celles de la sémantique dynamique.

3.1.3.2. Forme des règles de grammaire

La syntaxe de BASEL est entièrement définie par des règles de la forme :

non-terminal ::= phrase

où une phrase est une suite de terminaux et de non-terminaux, les seuls symboles métalinguistiques étant "::=", et le blanc pour la concaténation.

Les non-terminaux sont des suites de lettres minuscules éventuellement découpées en plusieurs mots par des tirets. Tous les non-terminaux contiennent plus d'une lettre. Les terminaux sont des suites de lettres minuscules soulignées, des lettres majuscules ou minuscules, des chiffres ou des caractères spéciaux.

La grammaire complète de BASEL telle qu'elle est définie par le traducteur utilise donc un formalisme beaucoup plus rudimentaire que celui qui a servi pour la description informelle. D'ailleurs, on remarquera que la grammaire utilisée dans le traducteur a besoin d'un plus grand nombre de non-terminaux pour définir le même langage.

Exemples de règles :

```
expression-composée ::= expression-composée-sérielle
expression-composée ::= expression-composée-collatéralle
expression-composée-sérielle ::= début-sériel corps-d-expression-composée
                                 fin-seriel
debut-seriel ::= (
fin-seriel ::= )
expression-composée-collatéralle ::= debut-collateral
                                     corps-d-expression-composée
                                     fin-collateral
debut-collateral ::= [
fin-collateral ::= ]
```

3.1.3.3. Forme de l'appel des procédures de la sémantique statique

A certaines des règles qui constituent la grammaire de BASEL est associé l'appel d'une procédure de la sémantique statique. Cet appel est simplement écrit, dans la ligne qui suit la règle considérée, sous la forme d'un ou plusieurs mots en lettres majuscules, dont l'ensemble constitue le nom de la procédure appelée.

EXEMPLES :

```
expression-composée ::= expression-composée-serielle
expression-composée ::= expression-composée-collateralle
expression-composée-serielle ::= debut-seriel corps-d-expression-composée
                                 fin-seriel
                    COMPOSEE SERIELLE
debut-seriel ::= (
                    OUVRIR BLOC
fin-seriel ::= )
                    FERMER BLOC
expression-composée-collateralle ::= debut-collateral
                                     corps-d-expression-composée
                                     fin-collateral
                    COMPOSEE COLLATERALLE
debut-collateral ::= [
                    OUVRIR BLOC
fin-collateral ::= ]
                    FERMER BLOC
```

3.1.3.4. Fonctionnement général du traducteur

Le traducteur est dirigé par la syntaxe.

Il analyse la représentation concrète d'un programme selon les règles de la grammaire. Le travail d'analyse s'effectue au cours d'une lecture de gauche à droite du texte du programme et l'ordre d'application des règles de grammaire est celui qui correspondrait à la construction de l'arbre syntaxique du programme dans l'ordre canonique par une analyse ascendante. Les informations sur lesquelles travaille l'analyseur propremnt dit ne sont connues que de lui seul et sont inaccessibles dans les autres composants du traducteur.

Lorsqu'une règle vient d'être appliquée, deux cas peuvent se présenter :

- Il n'y a pas de nom de procédure de la sémantique statique associé à la règle utilisée : l'analyse syntaxique se poursuit normalement.
- Il y a un nom de procédure de la sémantique statique associé à la règle utilisée : l'analyse syntaxique est suspendue, la procédure nommée est exécutée - ce qui peut entraîner éventuellement l'appel d'autres procédures - puis, lorsque l'exécution de cette procédure est terminée, l'analyse syntaxique est reprise normalement.

3.1.3.5. Les procédures de la sémantique statique

Les procédures de la sémantique statique sont décrites à l'aide d'un formalisme dont la plus grande partie est issue du langage AMBIT/G (Algebraic Manipulation By Identity Transformation/Graphical) développé par Carlos Christensen à Computer Associates [Cr68, He69] en 1968 et implémenté sur le TX-2 du Lincoln Laboratory par Rovner et Henderson [Ro69].

Dans les paragraphes qui suivent je donne donc d'abord une description rapide et informelle de ce qu'est AMBIT/G (une description formelle, de caractère mathématique, en est donnée en [He69]), puis je décris la façon dont j'ai modifié AMBIT/G pour qu'il puisse être utilisé dans la description des procédures de la sémantique statique, ainsi que dans celles de la sémantique dynamique.

3.1.3.6. Description d'AMBIT/G

AMBIT/G est un langage de nature bi-dimensionnelle, totalement indépendant d'une machine. Il opère sur des objets qui constituent les sommets d'un graphe orienté. Chaque instruction d'AMBIT/G effectue une vérification sur la configuration d'un sous-graphe du graphe qui contient les objets, et si la vérification est satisfaite, l'instruction peut éventuellement se poursuivre par une transformation de ce sous-graphe.

a. Notion de programme en AMBIT/G

Un programme AMBIT/G est composé de 4 parties :

- la définition des types ;
- la création des objets nommés ;
- la configuration initiale ;
- les instructions.

b. La définition des types

Dans cette partie du programme, sont définies les diverses sortes d'objets qui pourront constituer les sommets du graphe sur lequel les instructions effectueront leur travail.

AMBIT/G étant un langage bi-dimensionnel, c'est la forme du dessin qui sert à représenter un objet qui symbolise l'information sur son type. Ces formes doivent être des courbes fermées, comme :

Dans un programme en AMBIT/G, c'est l'utilisateur qui choisit lui-même les formes qu'il désire pour représenter les objets qu'il veut pouvoir manipuler, et par chaque forme il signifie des objets de nature différente. Par exemple,  représentera un index,  une position dans une pile,  une variable,  un élément d'une liste et  une valeur booléenne.

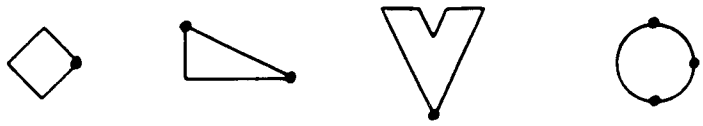
D'autre part, les objets étant les sommets d'un graphe orienté, chaque arc sert à représenter une relation entre deux objets.

D'un objet donné ne peuvent être issus qu'un nombre fixe et déterminé d'arcs, mais à tout objet peut aboutir un nombre quelconque d'arcs. Le nombre d'arcs qui partent d'un objet dépend de son type, et chaque arc a son origine représentée en un point particulier du contour de la forme qui représente l'objet, appelé point-origine.

La définition de chaque type d'objet spécifie donc :

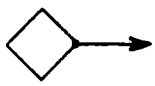
- la forme qui sert à représenter les objets de ce type ;
- les points-origines des arcs qui sont issus des objets de ce type.

Chaque point-origine est défini, lors de la définition des types, par un point marqué sur le contour de la forme qui symbolise le type. Exemples :



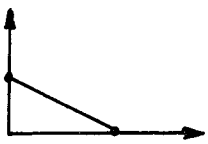
C'est l'utilisateur qui choisit lui-même le nombre et la position des points-origines des arcs qui peuvent partir de la représentation d'un objet, et à chaque arc issu d'un point-origine particulier il associe évidemment une signification propre.

EXEMPLES :

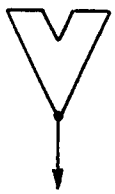


objet indiqué par cet index.

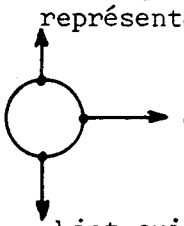
position précédente dans la pile.



objet qui représente la valeur contenue dans cette position de la pile.



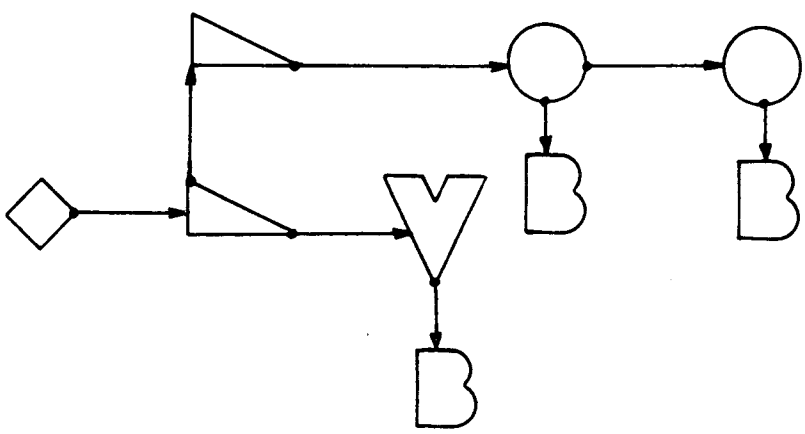
objet qui représente la valeur "retenue" par cette variable.
représentation du "nom" attaché à cet élément de liste.



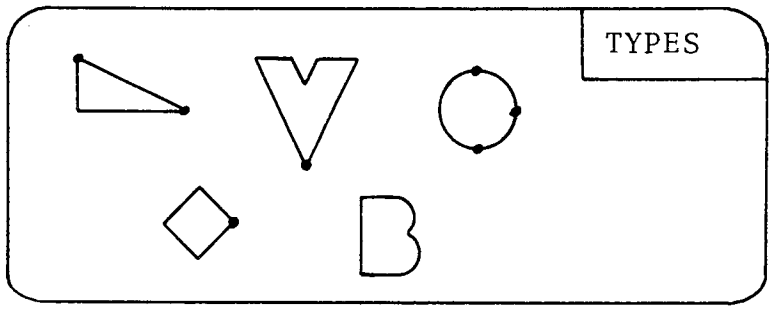
élément suivant dans la liste.

objet qui représente la valeur associée à cet élément de liste.

Un sous-graphe du graphe des objets pourrait ainsi avoir la configuration suivante :



La définition de tous les types d'objets pour un programme est elle-même inscrite dans une "boîte", à la façon de l'exemple suivant :



C'est cette "boîte" qui constitue la première partie du programme. Tous les objets manipulés par le programme sont de l'un des types ainsi définis et les instructions du programme font exclusivement usage de ces représentations.

Remarque : lors de la définition d'un type, "l'orientation" de la forme qui le symbolise est importante. Ainsi,  est différent de  et  est différent de  .

c. La création des objets nommés

Les instructions d'un programme pourront créer et détruire un nombre quelconque d'objets, avec création et destruction d'arcs correspondants. Ces objets seront toujours des objets anonymes. Cependant, certains objets sont permanents et indestructibles : ce sont les objets nommés.

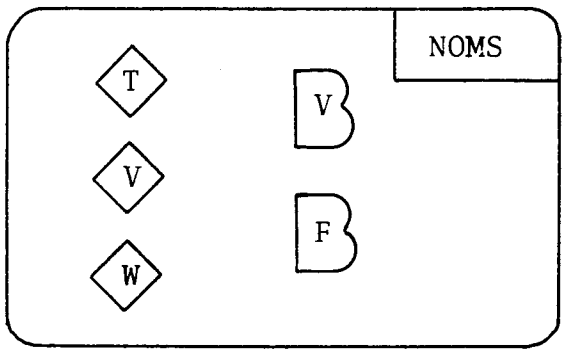
La création d'un objet nommé est faite dans la deuxième partie du programme, en inscrivant un identificateur ou une notation quelconque - le nom de l'objet - dans la représentation d'un objet du type désiré.

EXEMPLES :

Pour un type donné, il n'est permis de créer qu'un objet ayant un nom donné, mais un même nom peut être utilisé pour plusieurs objets, à condition qu'ils soient de types différents.

La création des objets nommés est inscrite dans une deuxième "boîte" du programme.

EXEMPLE :

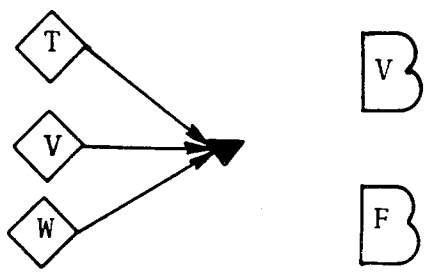


Comme on va le voir, ce sont les objets nommés, et eux seuls, qui permettent d'accéder aux autres objets du graphe.

d. La configuration initiale

Dans une troisième partie du programme est créée la configuration initiale du graphe à partir duquel les instructions du programme vont commencer leur travail. Cette partie du programme est optionnelle. Si elle n'est pas présente, le graphe initial est réduit à l'ensemble des objets nommés, et tous les arcs issus de ces objets aboutissent au "vide", qui est un noeud unique, conventionnellement représenté dans la définition formelle de BASEL par :

Donc, dans le cas de l'exemple précédent, le graphe initial serait :



(On peut remarquer que ce graphe n'est pas connexe : ceci est tout à fait possible en AMBIT/G et ne crée aucun problème particulier).

Par contre, si une configuration initiale est spécifiée, elle peut servir à la création d'un graphe quelconque, à condition qu'il respecte les règles de construction suivantes, qui sont celles auxquelles obéissent tous les graphes en AMBIT/G :

- seuls y sont présents des objets de types définis, c'est-à-dire représentés par des formes définies ;
- tout arc doit être issu d'un endroit sur le contour d'une forme, qui correspond à un point origine défini ;
- il ne peut y avoir plus d'un arc issu d'un point origine donné ;
- tout objet doit être accessible ;

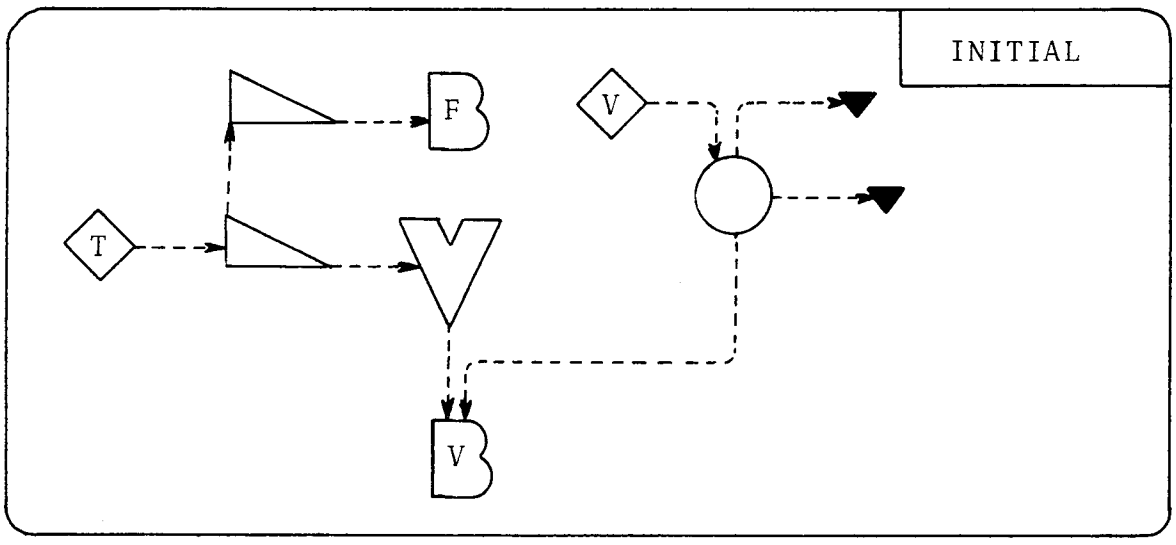
l'accessibilité d'un objet étant définie de la façon suivante :

- tout objet nommé est accessible ;
- tout objet situé à l'extrémité d'un arc issu d'un objet accessible est accessible.

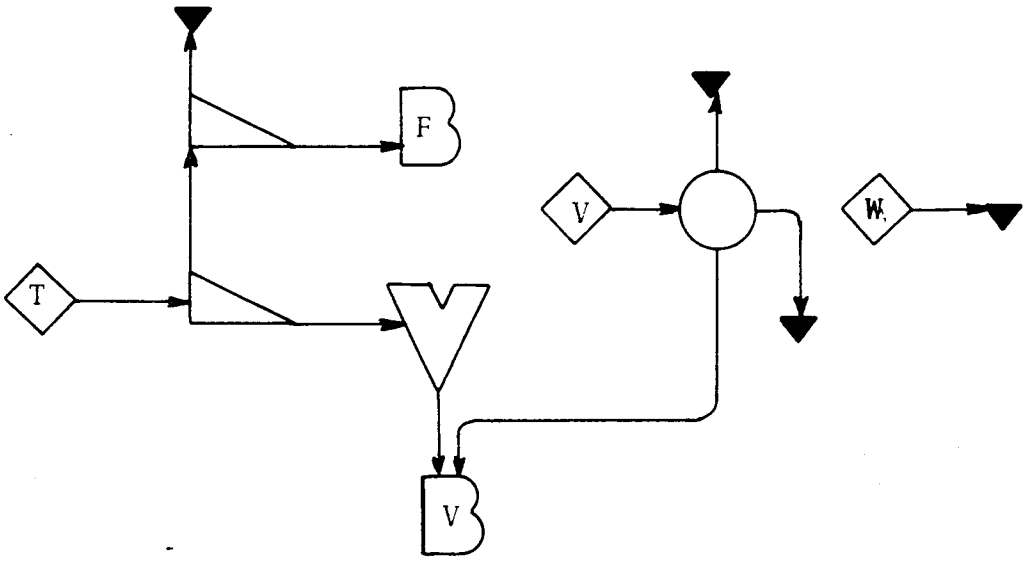
Pour créer la configuration initiale, les conventions suivantes sont utilisées, qui sont valables également pour les instructions du programme :

- la création d'un arc est provoquée par la représentation de cet arc dessiné en pointillés,
- la création d'un objet anonyme est provoquée par la représentation de la forme de cet objet, à laquelle n'aboutissent que des arcs en pointillés.

Exemple de création de configuration initiale :



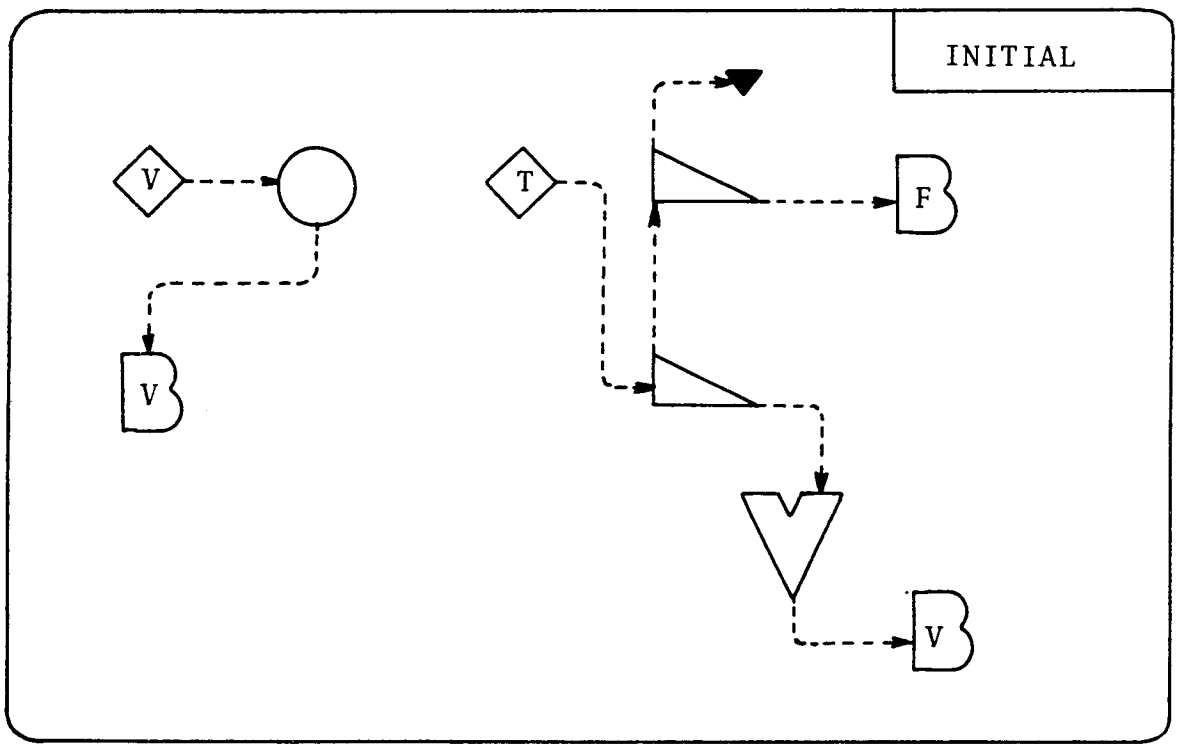
Ceci définit le graphe suivant, en considérant que l'on a les objets nommés créés plus haut :



Sur cet exemple, on peut faire quelques remarques qui sont également valables pour les instructions du programme :

- lorsqu'un nouvel objet anonyme est créé, tous les arcs issus de cet objet et qui ne sont par représentés aboutissent au "vide" ;
- tout objet unique (c'est-à-dire le "vide", ou un objet nommé) peut être représenté graphiquement plusieurs fois dans la configuration initiale - ou dans une même instruction - : il s'agit toujours du même objet.
- la disposition des formes et des arcs n'a aucune importance. Seules comptent les formes elles-mêmes (avec leur "orientation") et les points-origines des arcs représentés. L'endroit du contour d'une forme auquel aboutit un arc n'a pas non plus d'importance, car un arc aboutit à l'objet représenté par la forme.

Ainsi, la même configuration initiale que ci-dessous serait obtenue par :



e. Les instructions

Dans la dernière partie d'un programme, celle qui contient les instructions, on distingue deux aspects :

- le contenu de chaque instruction proprement dite ;
- le passage du contrôle d'une instruction à une autre.

e.1. Contenu d'une instruction

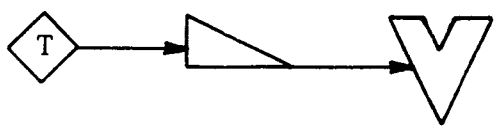
Une instruction commence par effectuer une reconnaissance de forme qui se réduit à une simple vérification sur la configuration d'un sous-graphe du graphe des objets et, si cette vérification est satisfaite, l'instruction peut provoquer une transformation de ce sous-graphe.

Dans une instruction, on distingue donc deux parties :

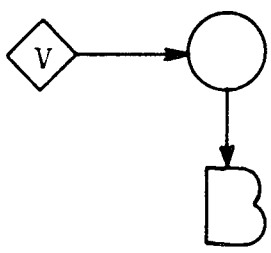
- une vérification ;
- une transformation.

La vérification est exprimée simplement, dans l'instruction, par la représentation d'un sous-graphe dont les sommets, tous de types définis, doivent tous être accessibles. Si, à l'instant où est exécutée l'instruction, le sous-graphe représenté est bien un sous-graphe du graphe total des objets, la vérification est satisfaite. Sinon, elle n'est pas satisfaite.

Par exemple, le graphe étant celui construit par la configuration initiale donnée plus haut, une vérification satisfaite peut être exprimée par :



ou par :



où l'objet de type représenté exige indifféremment la présence de

, de

ou d'un objet quelconque de ce type.

Une vérification non satisfaite serait par exemple :

ou encore :

Etant donné la façon dont est spécifiée une vérification, on voit que le sous-graphe qui est "reconnu" de cette façon, s'il existe, est forcément unique : ceci est dû à la règle d'accessibilité des sommets et à l'unicité des objets nommés.

Lorsque la partie "vérification" d'une instruction a été effectuée, trois cas peuvent se présenter :

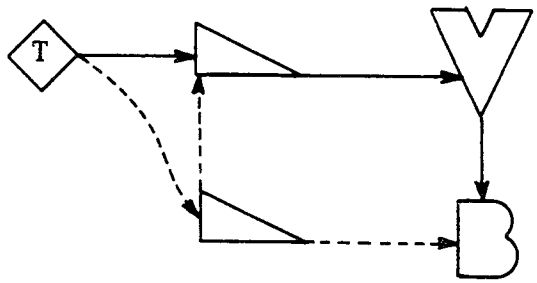
- si la vérification n'est pas satisfaite, l'instruction est terminée, et elle résulte en un ECHEC.
- si la vérification est satisfaite, et s'il n'y a pas de partie "transformation", l'instruction est terminée, et elle résulte en un SUCCES.
- si la vérification est satisfaite et s'il y a une partie "transformation", celle-ci est exécutée, à la suite de quoi l'instruction est terminée et résulte en un SUCCES.

La spécification des transformations est "superposée" à la représentation du sous-graphe qui définit la vérification. Les transformations elles-mêmes sont exprimées selon des conventions analogues à celles qui ont servi à la création de la configuration initiale :

- Il est possible de remplacer un arc existant par un autre arc en représentant, en pointillés, cet autre arc issu du même point origine que l'arc existant.

- Il est possible de créer des objets anonymes, en représentant leur forme et en créant des arcs qui les rendent accessibles.
- Tout objet anonyme rendu inaccessible par les transformations est supprimé du graphe des objets, ainsi que les arcs qui sont issus de cet objet.

Exemple d'instruction avec transformation, à partir de la configuration initiale définie plus haut :



Ceci est exécuté de la façon suivante :

"Si l'arc issu de l'objet T de type ◇ aboutit à un objet de type ◺ dont l'arc de droite aboutit à un objet de type ▽ qui lui-même a son arc unique qui aboutit à un ▯ quelconque, alors l'instruction est un SUCCES, sinon c'est un ECHEC. De plus, si l'instruction est un SUCCES, il est créé un nouvel objet de type ◺ dont l'arc de droite aboutit au ▯ trouvé précédemment et dont l'arc supérieur aboutit au ◺ situé à l'extrémité de l'arc issu de ◇T. Enfin, l'arc issu de ◇T est remplacé par un arc aboutissant au nouveau ◺ créé".

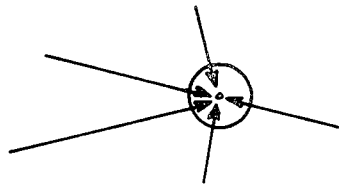
Mais, dans l'esprit de l'utilisateur, cela signifie peut être simplement :

"Si le sommet de la pile, indiqué par T, contient une variable à valeur booléenne, empiler cette valeur".

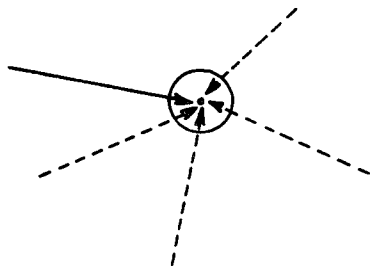
On remarque donc la concision avec laquelle la représentation graphique traduit les intentions de l'utilisateur.

Enfin, il faut mentionner une abréviation que j'ai introduite à l'occasion de l'utilisation d'AMBIT/G pour la définition formelle de BASEL :

- Si l'on veut, dans la vérification, exprimer que plusieurs arcs aboutissent au même objet, sans préciser nécessairement de quel objet ni de quel type d'objet il s'agit, les extrémités de ces arcs sont représentés groupées de la façon suivante :

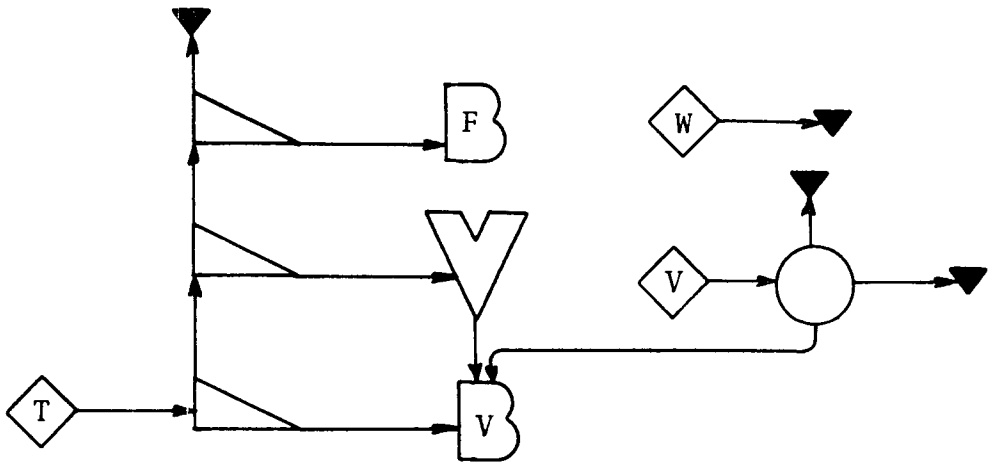


- Si l'on veut, dans la transformation, faire aboutir des arcs nouveaux à l'objet auquel aboutit déjà au moins un arc existant, sans préciser nécessairement de quel objet ni de quel type d'objet il s'agit, l'extrémité de l'arc existant et celles des arcs nouveaux sont représentées groupées de la façon suivante :

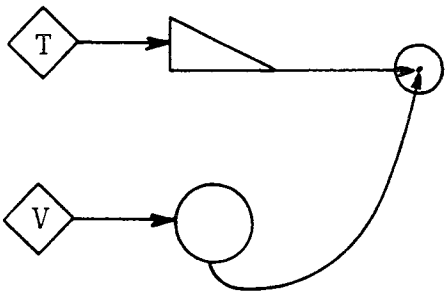


EXEMPLE :

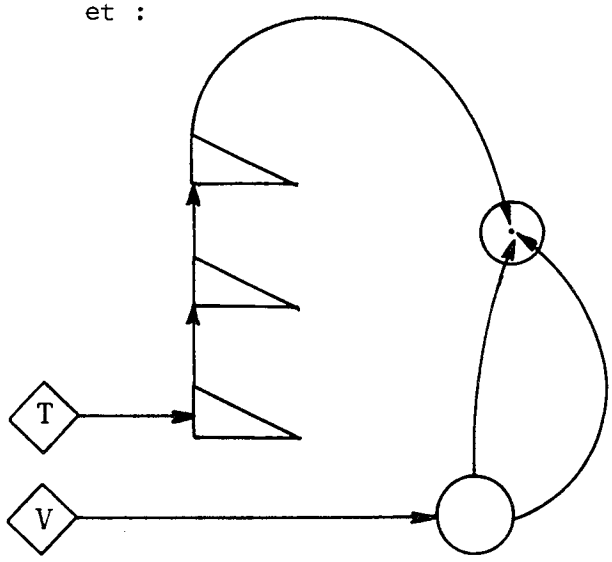
Après exécution de l'instruction de l'exemple précédent, la configuration du graphe complet est :



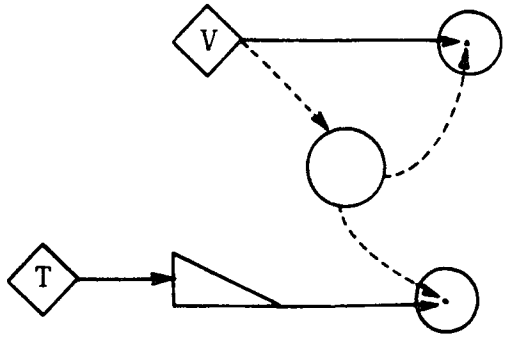
Les vérifications suivantes seront alors satisfaites :



et :



Et une instruction avec transformation pourrait être :

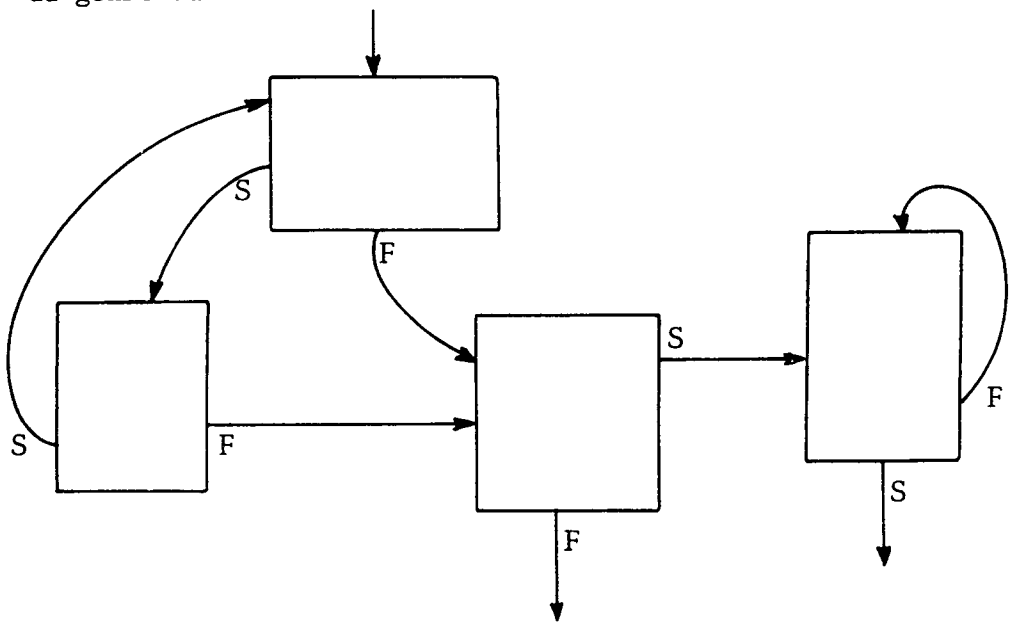


e.2. Passage du contrôle

Le passage du contrôle entre les instructions d'un programme est lui aussi représenté de façon graphique.

Comme on l'a vu plus haut, une instruction peut résulter en un ECHEC ou en un SUCCES. On inscrira alors la spécification de la vérification et celle de la transformation éventuelle demandées par une déclaration dans une "boîte" rectangulaire avec deux sorties : une sortie "S" pour SUCCES et une sortie "F" pour ECHEC (Failure). Chaque sortie sera représentée par une flèche partant d'un bord de la "boîte" qui contient l'instruction, portant la lettre S ou la lettre F selon la sortie représentée, et aboutissant a une autre "boîte" (ou la même) contenant une instruction.

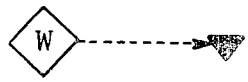
L'ensemble de la partie instruction d'un programme a donc une allure du genre suivant :



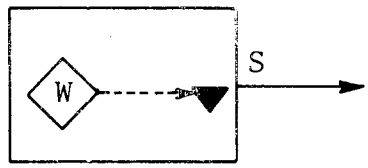
où chaque "boîte" contient une instruction.

La flèche "issue de nulle part" indique la première instruction à exécuter dans le programme, et les flèches n'aboutissant nulle part provoquent la fin de l'exécution du programme.

D'autre part, certaines instructions peuvent n'avoir qu'une seule flèche (en général "S") issue de la boîte qui les contient : c'est que l'autre flèche (en général "F") est sans signification. Par exemple, une instruction comme :



est toujours un SUCCES, et provoque toujours la transformation indiquée. Donc, elle sera toujours mise sous la forme :

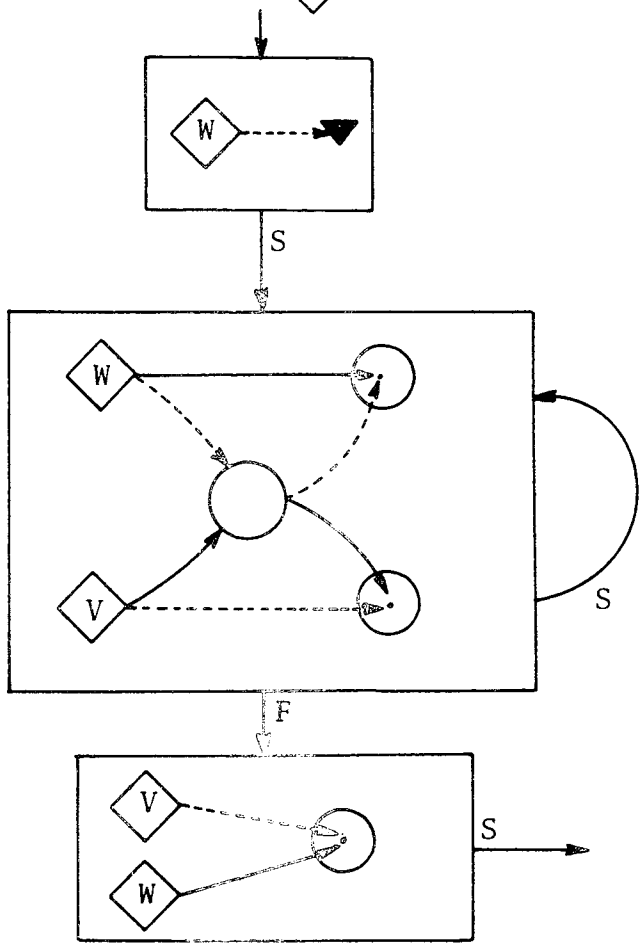


Exemple de partie instruction :

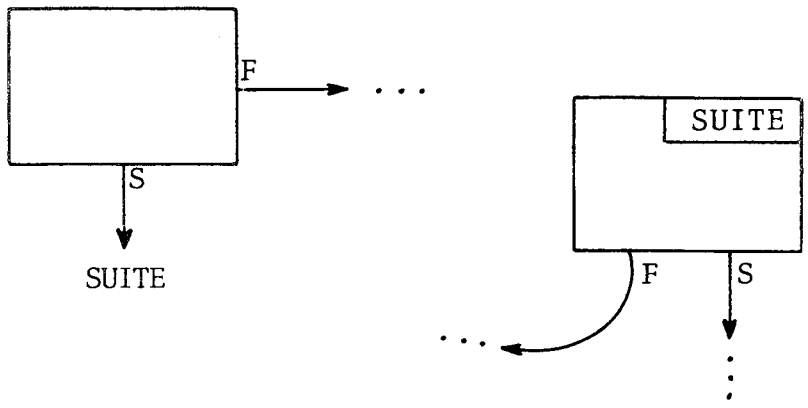
Si 

indique le début d'une liste de  , une façon de renverser l'ordre des éléments de cette liste et d'en indiquer toujours le début par 

est :



Enfin, il peut arriver, pour des raisons d'encombrement ou d'organisation du programme, qu'il soit peu pratique ou même impossible de tracer une flèche d'une instruction à une autre. On peut alors donner des noms à un certain nombre d'instructions et s'y référer en indiquant le nom voulu à l'extrémité de la flèche qui devrait aboutir à l'une de ces instructions :



3.1.3.7. Intégration d'AMBIT/G dans la définition formelle de BASEL

Les procédures de la sémantique statique utilisent AMBIT/G pour construire et conserver les informations qui leur sont nécessaires (modes, déclarations, instructions) dans un graphe d'objets d'AMBIT/G : lorsque le programme concret a été complètement analysé, ce graphe constitue la représentation abstraite du programme. Cette représentation abstraite joue alors le rôle de "configuration initiale" pour les procédures de la sémantique dynamique qui constituent l'interpréteur.

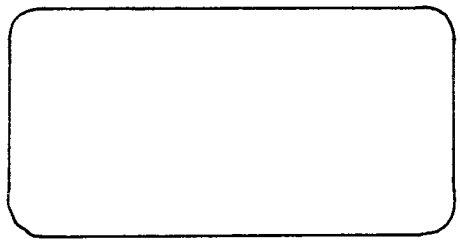
Pour réaliser cela, j'ai défini une structure de procédures afin d'accueillir de façon systématique les instructions d'AMBIT/G.

a. Définition des procédures

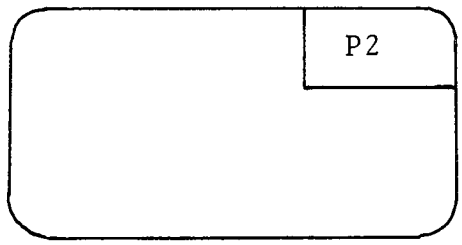
Dans ce système, une procédure a plusieurs caractéristiques :

- son nom,
- ses paramètres,
- ses objets nommés locaux,
- ses instructions,
- son point d'entrée,
- ses sorties.

Pour une procédure donnée, toutes ces caractéristiques sont spécifiées graphiquement à l'intérieur d'une "boîte" de la forme :

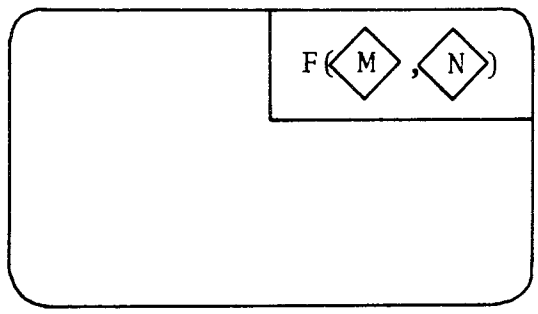


Le nom de la procédure est constitué d'un ou plusieurs mots écrits en majuscule et contenant éventuellement des chiffres. Il est écrit dans un cadre situé en haut et à droite de la boîte qui contient la procédure :



Si la procédure a des paramètres, la liste des paramètres formels est définie à la suite du nom. Elle est mise entre parenthèses et les paramètres sont séparés par des virgules. Chaque paramètre formel a un nom et un type (spécifié par sa forme) : la définition d'un paramètre formel est donc analogue à celle d'un objet nommé qui n'existe et n'est connu qu'à l'intérieur de la procédure :

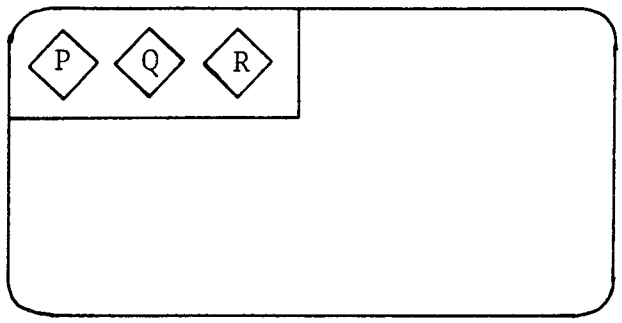
Exemple :



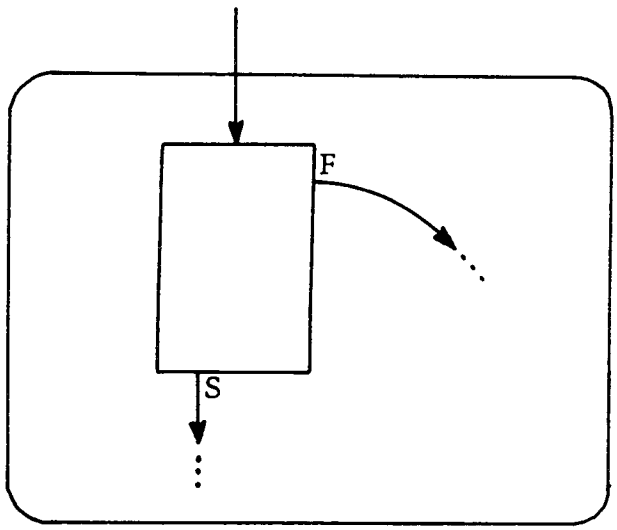
En plus de ses paramètres, une procédure peut faire usage d'autres objets nommés : certains d'entre-eux sont globaux à l'ensemble de toutes les procédures et les autres sont locaux à cette procédure. Comme celle des paramètres formels, l'existence des objets nommés locaux est limitée à l'intérieur de la procédure, et ils sont inconnus des autres procédures.

Les objets nommés locaux sont définis par une liste représentée dans un cadre situé en haut et à gauche de la boîte qui contient la procédure :

Exemple :



Les instructions qui constituent le corps d'une procédure sont des instructions AMBIT/G (ou des appels : voir plus loin) entre lesquelles le contrôle est défini comme pour la partie instruction d'un programme AMBIT/G normal. Le point d'entrée de la procédure est indiqué par une flèche qui traverse le bord supérieur de la boîte et aboutit à la première instruction qui sera exécutée lors d'un appel de cette procédure, après liaison éventuelle des paramètres et création des objets nommés locaux :

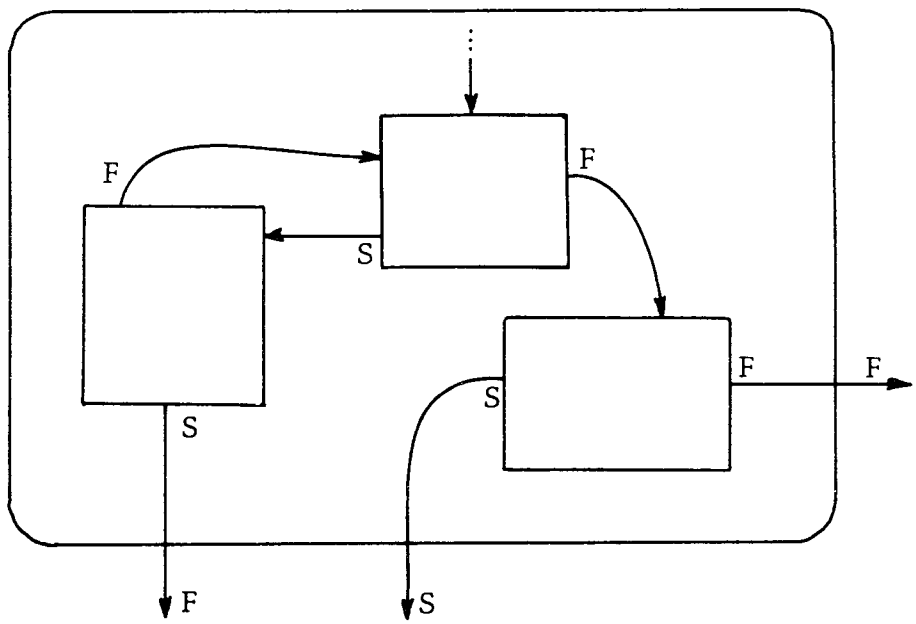


En général, une procédure a deux sorties : une sortie SUCCES et une sortie ECHEC. Elles sont respectivement représentées par des flèches, qui prolongent les flèches indiquant la fin des instructions, et qui sortent de la boîte qui contient la procédure. L'extrémité de ces flèches porte l'indication "S" ou "F", à l'extérieur de la boîte de la procédure, pour marquer qu'elles représentent la fin de la procédure par SUCCES ou par ECHEC. (L'origine de ces flèches, sur une instruction de la procédure, porte elle

aussi une indication "S" ou "F", comme toute flèche issue d'une instruction. Cependant, une flèche de sortie de la procédure commençant, par exemple, avec un F peut fort bien porter un S à son extrémité si elle est considérée comme représentant un SUCCES de la procédure bien qu'il s'agisse d'un ECHEC de l'instruction d'où elle part).

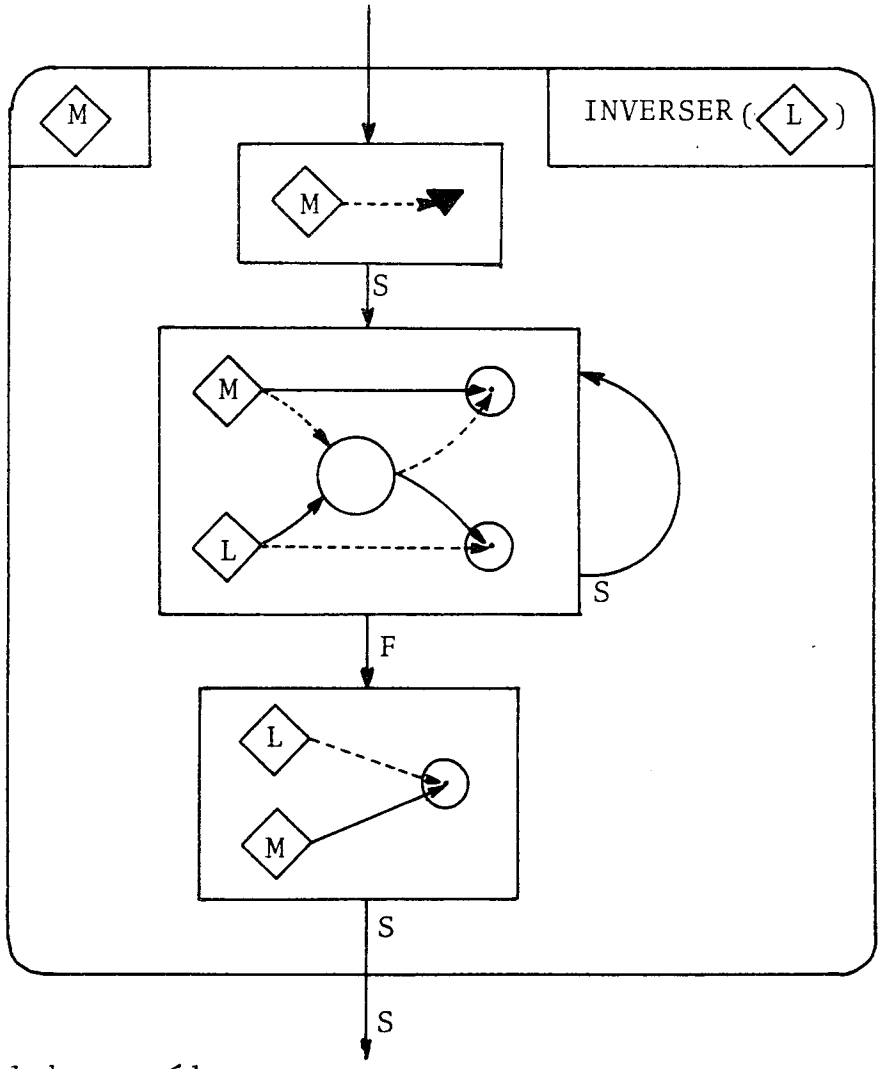
D'autre part, il peut y avoir plus de deux flèches sortant de la procédure : toutes celles qui portent un S à leur extrémité sont considérées comme une sortie unique avec SUCCES de la procédure, et toutes celles qui portent un F à leur extrémité sont considérées comme une sortie unique avec ECHEC de la procédure :

Exemple :



Enfin, comme une instruction, une procédure peut n'avoir qu'un seul type de sortie, en général "S".

Exemple de définition de procédure :



b. Appel des procédures

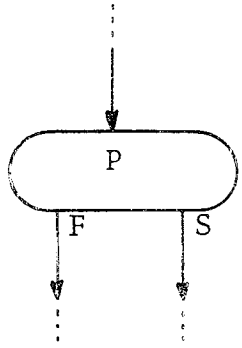
Dans la définition formelle de BASEL, il y a deux formes d'appels de procédures :

- les appels associés à l'utilisation d'une règle de grammaire ;
- les appels exprimés à l'intérieur d'une procédure.

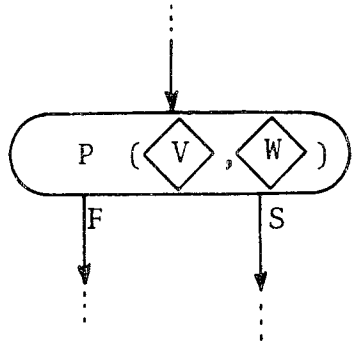
On a vu plus haut comment sont spécifiés les appels associés aux règles de grammaire. Il s'agit toujours d'appels de procédures sans paramètres et n'ayant pas de sortie ECHEC.

A l'intérieur d'une procédure, l'appel d'une autre procédure P, ou de cette procédure elle-même car la récursivité des appels est possible, a le statut d'une instruction et est spécifié dans une boîte "ovale" avec

sorties F et S si P a ces deux types de sorties, ou seulement S si P n'a pas de sortie ECHEC :



Si P a des paramètres, la liste des paramètres effectifs est inscrite dans l'appel :



Les paramètres effectifs sont toujours des objets nommés, qui doivent être du même type que les paramètres formels de la procédure.

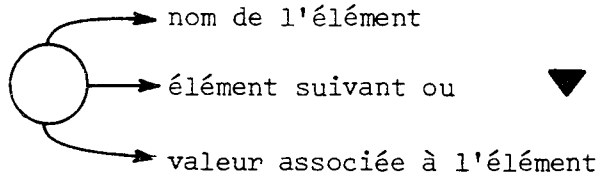
L'exécution d'un appel se déroule de la façon suivante :

- les paramètres effectifs sont liés par nom aux paramètres formels.
- les objets nommés locaux sont créés.
- l'exécution des instructions commence par l'instruction du point d'entrée, et se poursuit jusqu'à une sortie.
- si la sortie utilisée porte un S, l'appel est un SUCCES, sinon c'est un ECHEC.

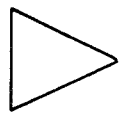
c. Exemple

Objet des procédures de cet exemple : associer une valeur à un élément d'une liste repéré par son nom.

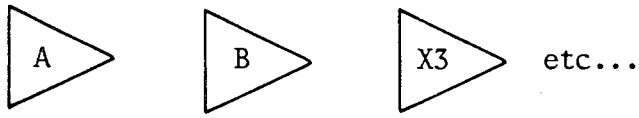
Les éléments d'une liste ont la forme :



Le nom d'un élément est un nouveau type d'objets, que l'on représentera par la forme :

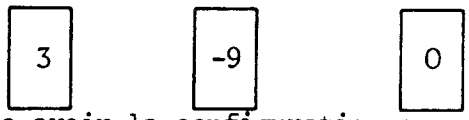


Aucun arc n'est issu de ces objets et ce sont tous des objets nommés :

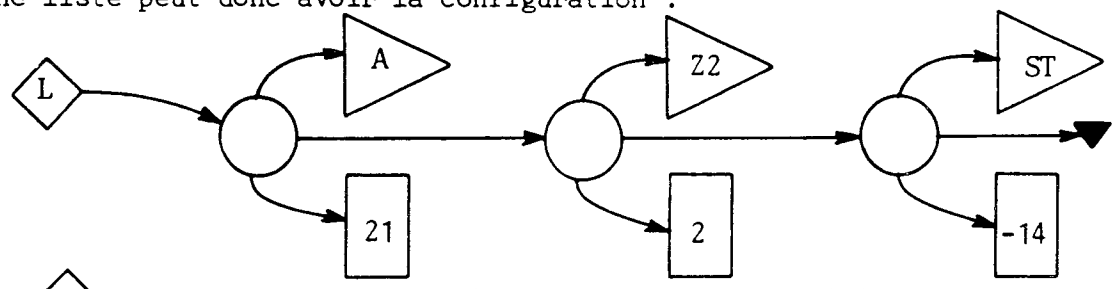


La valeur d'un élément est un entier. On représentera les entiers par la forme :

Les entiers sont tous des objets nommés :



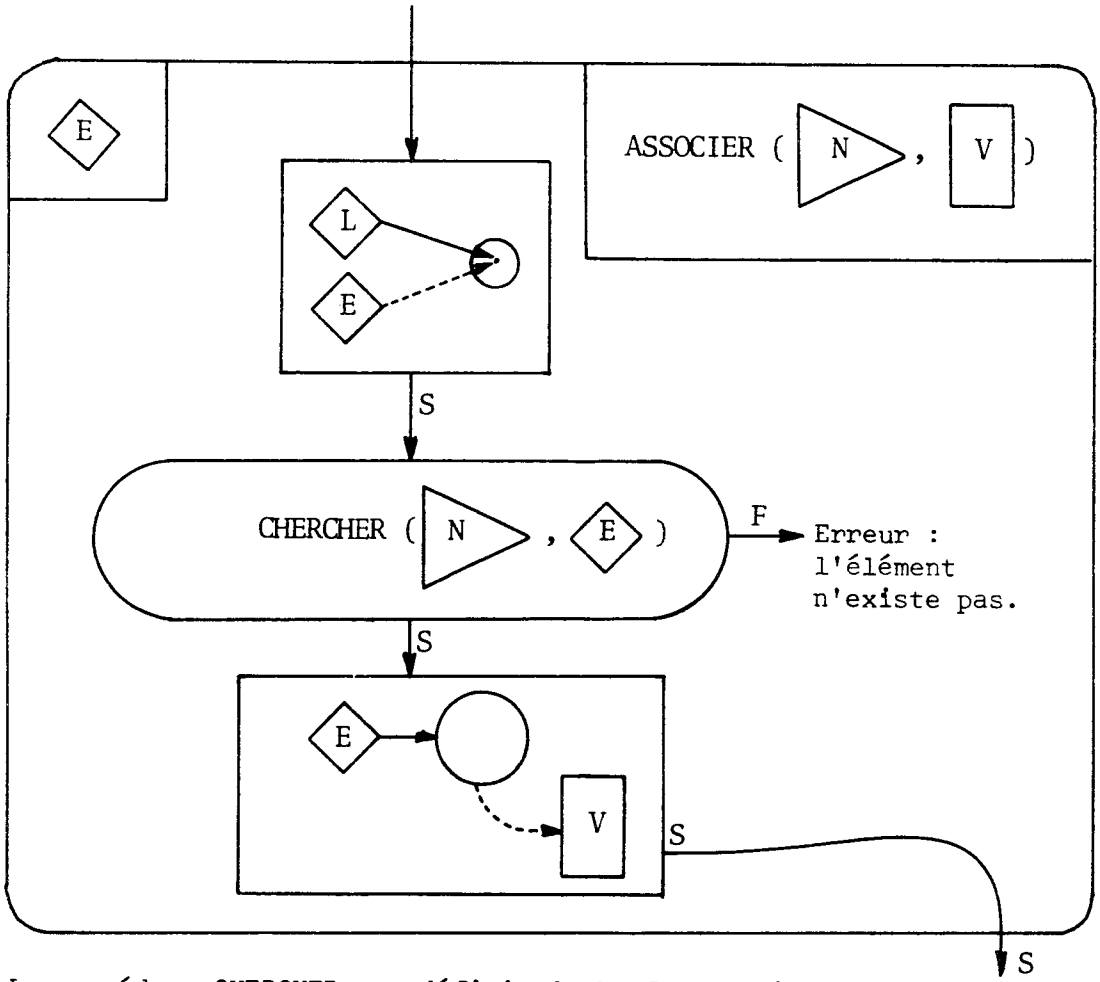
Une liste peut donc avoir la configuration :



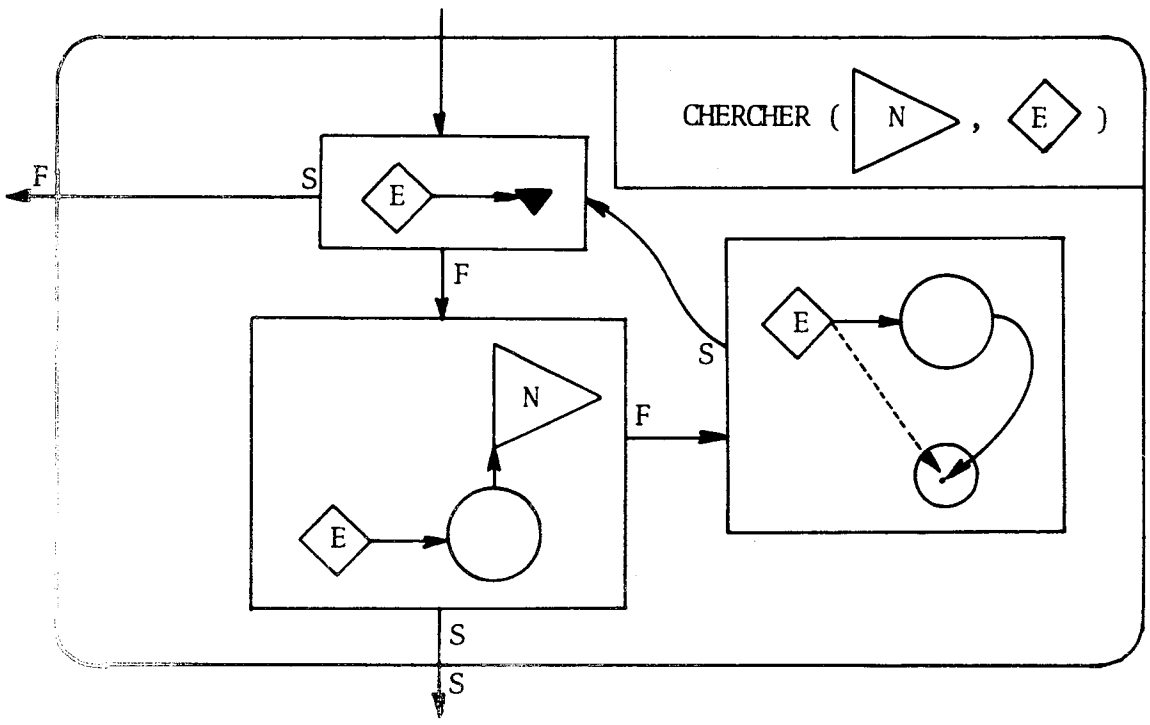
où 

est un index qui sert à indiquer le premier élément d'une liste.

L'association d'une valeur à un élément peut alors être accomplie par la procédure suivante, qui prend comme paramètre un nom d'élément et la valeur à associer à l'élément qui porte ce nom :



La procédure CHERCHER est définie de la façon suivante :



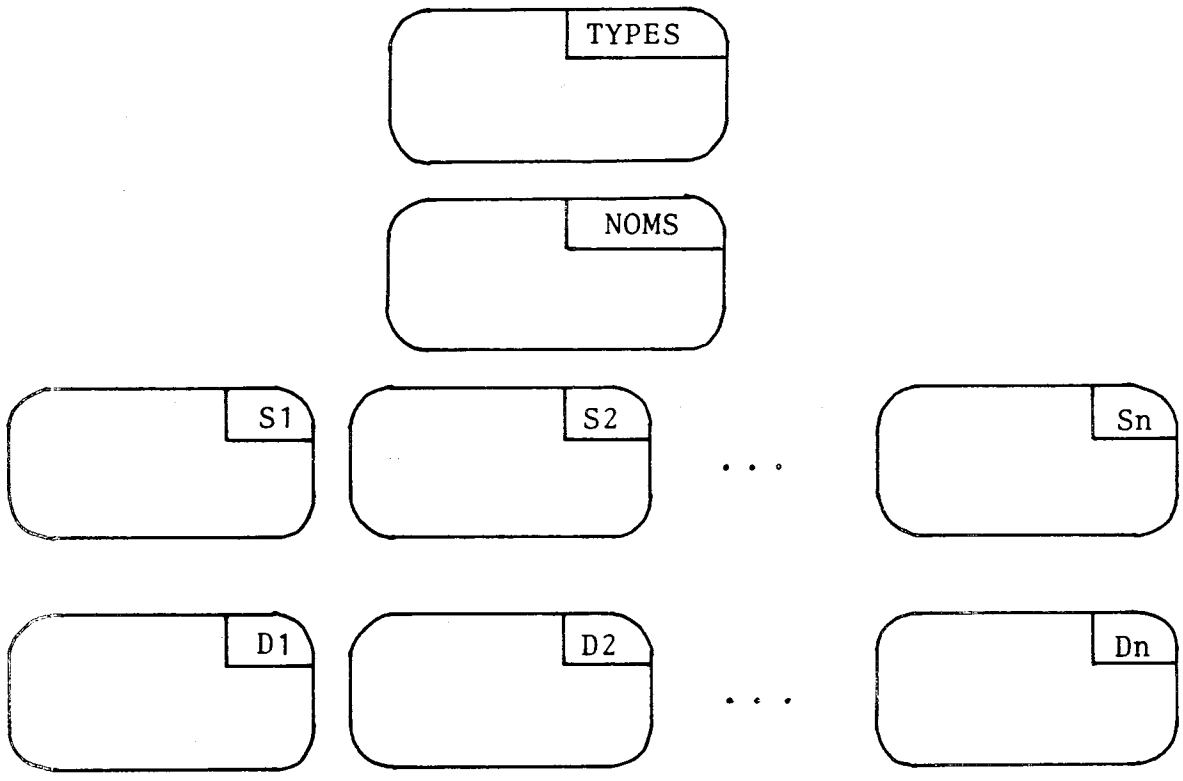
3.1.3.8. Vue d'ensemble du formalisme de la définition de BASEL

Pour résumer brièvement ce qui vient d'être dit, le traducteur qui définit la syntaxe et la sémantique statique de BASEL est constitué comme suit :

- un ensemble de règles hors-contexte. A certaines règles est associé l'appel d'une procédure, sans paramètre, de la sémantique statique.
- un ensemble de procédures dont le rôle est de définir la sémantique statique et de construire la représentation abstraite d'un programme. Ces procédures sont exprimées en AMBIT/G et la représentation abstraite qu'elles construisent est un graphe d'objets d'AMBIT/G.

Les types d'objets utilisés sont définis au préalable, comme pour un programme AMBIT/G normal, ainsi que les objets nommés globaux. Il n'y a pas de définition de configuration initiale : tous les arcs issus des objets nommés globaux créés aboutissent au vide au moment où commence la traduction.

La partie AMBIT/G de la définition a donc la forme :



où les Si sont les procédures de la sémantique statique et les Dj celles de la sémantique dynamique.

Les procédures de la sémantique dynamique constituent, à elles seules, tout l'interpréteur. Ces procédures sont construites selon les mêmes conventions que celles de la sémantique statique. Elles travaillent sur les mêmes types d'objets et les objets nommés globaux qu'elles utilisent sont définis en même temps que ceux de la sémantique statique.

"Statiquement", dans la partie AMBIT/G de la définition, il y a une portée des objets nommés. Lorsqu'un objet nommé ⟨N⟩ est utilisé dans une instruction située dans une procédure P (Si ou Dj), il s'agit d'un objet nommé local ou d'un paramètre de P si ⟨N⟩ est défini comme tel dans P, sinon il s'agit de l'objet nommé global ⟨N⟩, qui doit être défini. D'autre part, les procédures sont toutes externes les unes aux autres, et toute procédure peut être appelée de n'importe quelle autre procédure (elles sont toutes "au même niveau", aussi bien celles de la sémantique statique que celles de la sémantique dynamique).

"Dynamiquement", les objets nommés globaux sont créés une fois pour toute au début du processus de traduction, tandis que les objets nommés locaux d'une procédure P (Si ou Dj) sont créés chaque fois que P est appelée et détruits quand l'exécution de l'appel est terminée.

3.1.3.9. Utilisation d'AMBIT/G dans la définition de BASEL

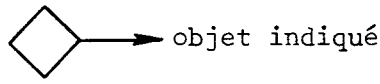
a. Types des objets

Pour la définition de BASEL, les objets qu'il faut pouvoir représenter appartiennent aux trois catégories suivantes :

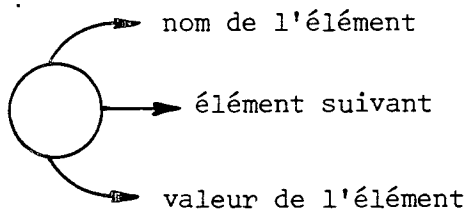
- objets d'utilité générale employés par la sémantique statique, la traduction et la sémantique dynamique ;
- objets représentant les modes et objets représentant les valeurs ;
- objets servant à représenter la structure et les instructions d'un programme.

b. Objets d'utilité générale

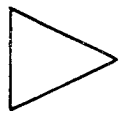
- les index :



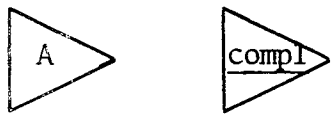
- les éléments de listes :



- les noms :



Les noms sont tous des objets nommés. Exemples :



c. Représentation des modes

- les modes de base :

ent :

reel :

bool :

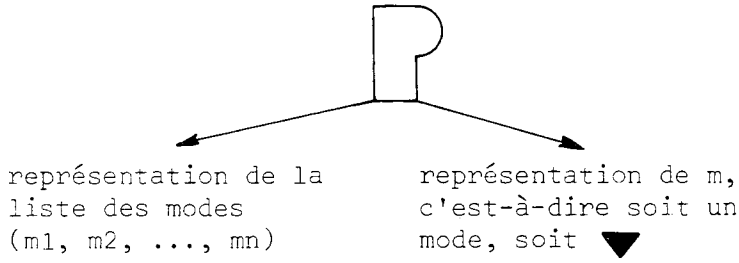
car :

Ce sont des objets nommés.

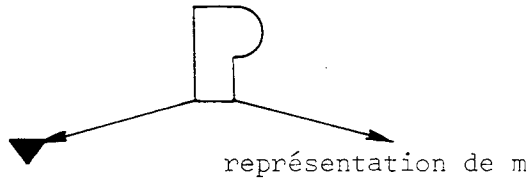
- le mode var m :

représentation de m

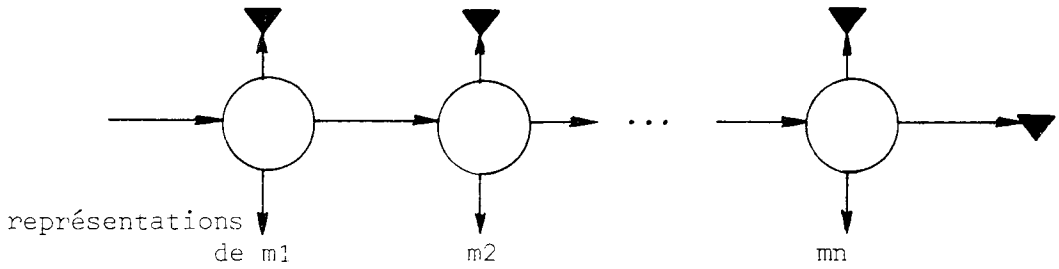
- le mode proc(m1, m2, ..., mn) m (où m est soit un mode, soit rien) :



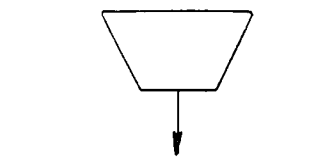
- le mode proc m :



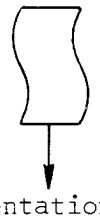
- Représentation d'une liste de mode (m1, m2, ..., mn) :



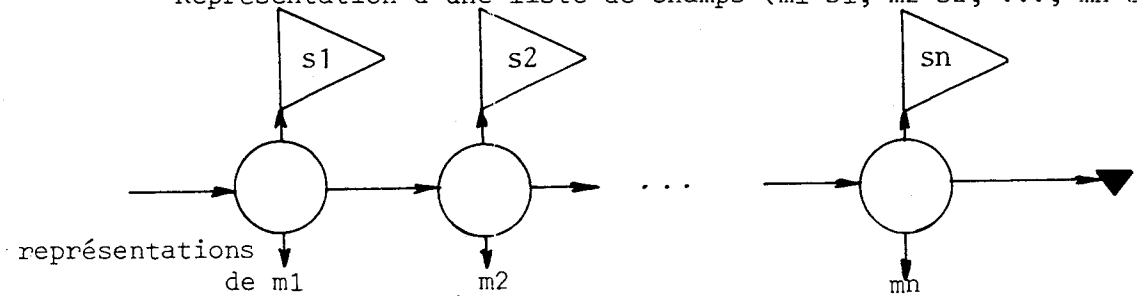
- Le mode tuple (m1, m2, ..., mn) :



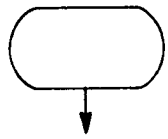
- le mode struct (m1 s1, m2 s2, ..., mn sn) :



- Représentation d'une liste de champs (m1 s1, m2 s2, ..., mn sn) :

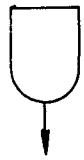


- Le mode seq m :



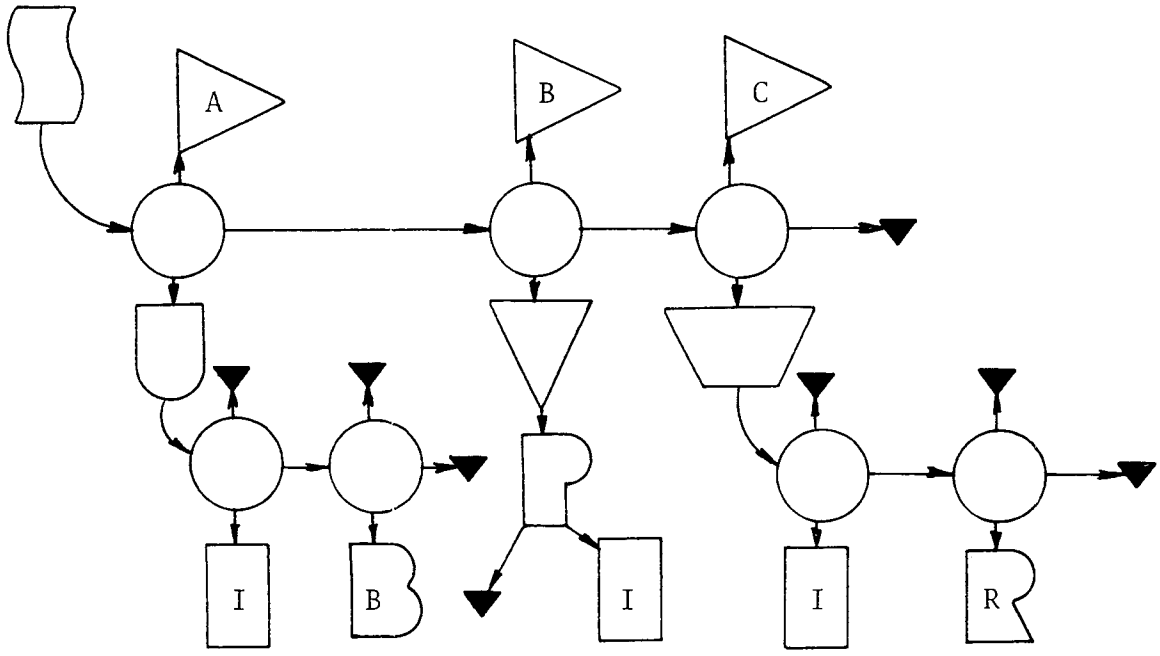
représentation de m

- Le mode union(m1, m2, ..., mn) :



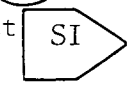
représentation de la
liste des modes
(m1, m2, ..., mn)

Ainsi le mode struct (union (ent, bool) A, var proc ent B, tuple (ent, reel) C) est représenté par :



D'autre part, deux autres objets sont nécessaires au traitement des modes dans la sémantique statique. Ce sont :

, qui sert aux déclarations de générique

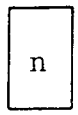
et 

, qui sert à indiquer qu'une branche d'instructions est sans issue.

Ces deux objets sont les seuls de leur type respectif.

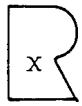
d. Représentations des valeurs et des objets de la sémantique dynamique

- Les valeurs des modes de base :

- Les entiers : 

où n est une notation d'entier.

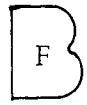
Exemple : 

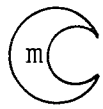
- Les réels : 

où x est une notation de réel.

Exemple : 

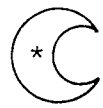
- Les booléens : 

et 

- Les caractères : 

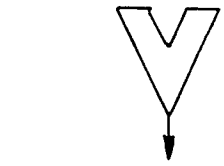
où m est une marque.

Exemples : 



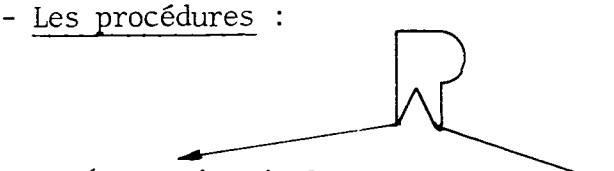
Les valeurs des modes de base sont donc toutes représentées par des objets nommés. La définition formelle de BASEL suppose ainsi la préexistence de tous les entiers et de tous les réels qu'un programme peut construire.

- Les variables :



représentation de la
valeur retenue par
la variable

- Les procédures :



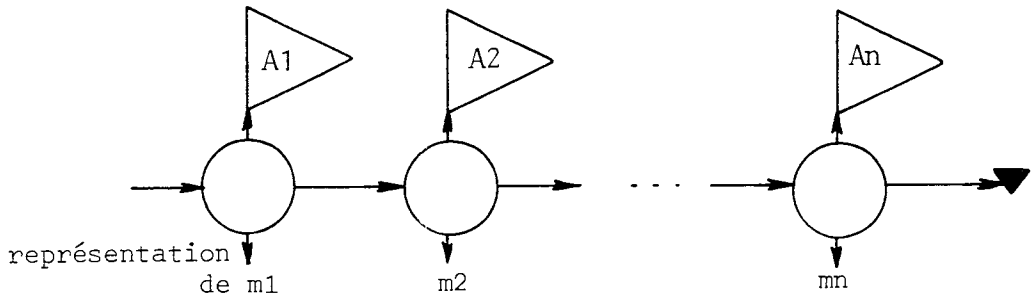
représentation de la
liste des paramètres
formels

représentation de la
partie instruction

- Représentation d'une liste de paramètres formels :

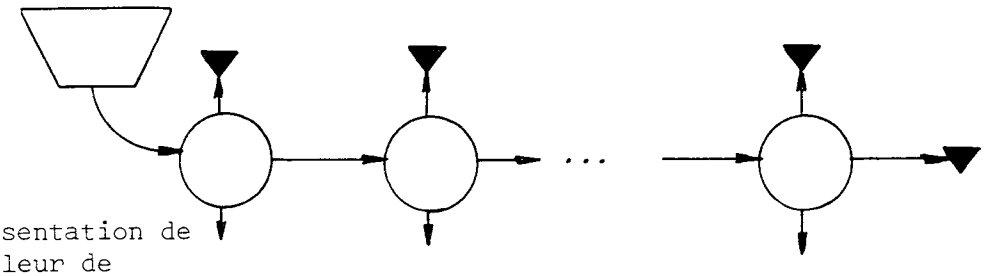
. si elle est vide : ▼

. si elle est de la forme A1 est m1, A2 est m2, ..., An est mn :



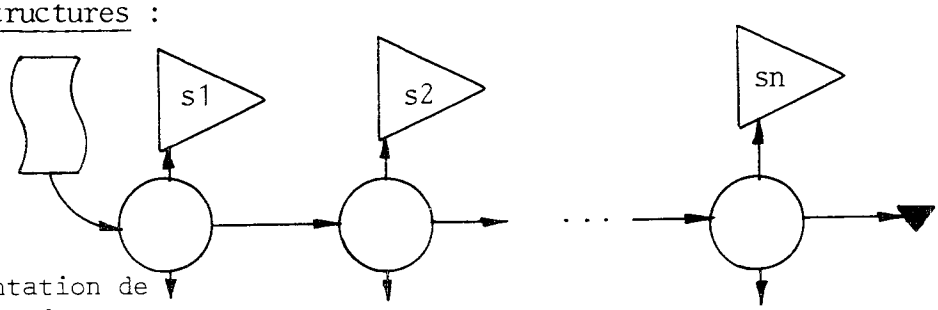
- Représentation d'une partie instruction :
(voir la représentation de la structure et des instructions d'un programme, paragraphe suivant).

- Les tuples :



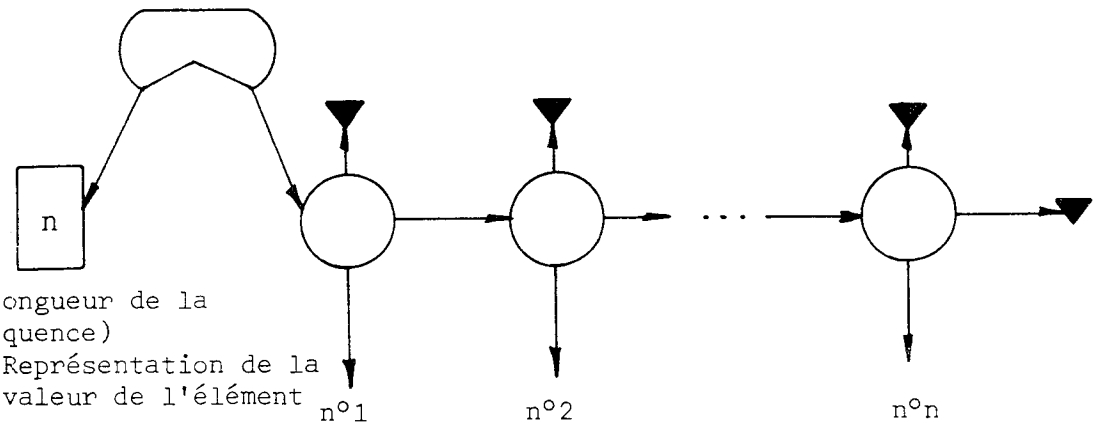
Représentation de la valeur de l'élément n°1 n°2 n°n

- Les structures :



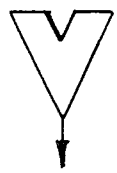
Représentation de la valeur du champ n°1 n°2 n°n

- Les séquences :



(longueur de la séquence)
Représentation de la valeur de l'élément n°1 n°2 n°n

où la valeur de chaque élément est de la forme :

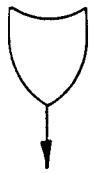


valeur retenue par la variable qui constitue l'élément

- Les unions :

Pas de représentation particulière, toute valeur ayant une représentation qui indique quel est son mode. Cependant, l'information sur le mode d'une valeur ne sera utilisée dynamiquement que lors de l'exécution d'une déclaration conditionnelle.

D'autre part, une origine est représentée par :



valeur associée

Enfin, la représentation de la valeur indéfinie d'un mode donné est simplement constituée de la représentation même de ce mode.

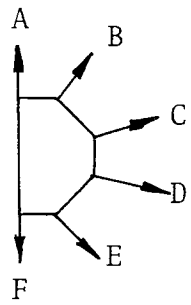
e. Représentation de la structure et des instructions d'un programme

Les procédures de la sémantique statique utilisent deux piles :

- une pile où est représentée l'imbrication statique courante des blocs et des procédures, et qui contient toutes les informations sur les contrôles ;
- une pile qui sert à construire la représentation abstraite du programme.

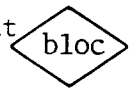
- Imbrication des blocs

Le sommet de la pile qui représente l'imbrication statique des blocs est indiqué par l'index ◇B◇ . Chaque position dans cette pile est représentative d'un bloc ou d'une procédure et a la forme :



où :

- A est soit ▼ soit la position précédente dans la pile ;

- B permet de distinguer entre bloc normal et procédure ; B est un "losange" qui est soit 

soit 

.

- C est la liste des déclarations simples. C contient aussi les déclarations de génériques et, après traitement de toute la partie déclaration, contient également les déclarations additionnelles ;

- D est la liste des déclarations conditionnelles ;

- E est la liste des déclarations de mode ;

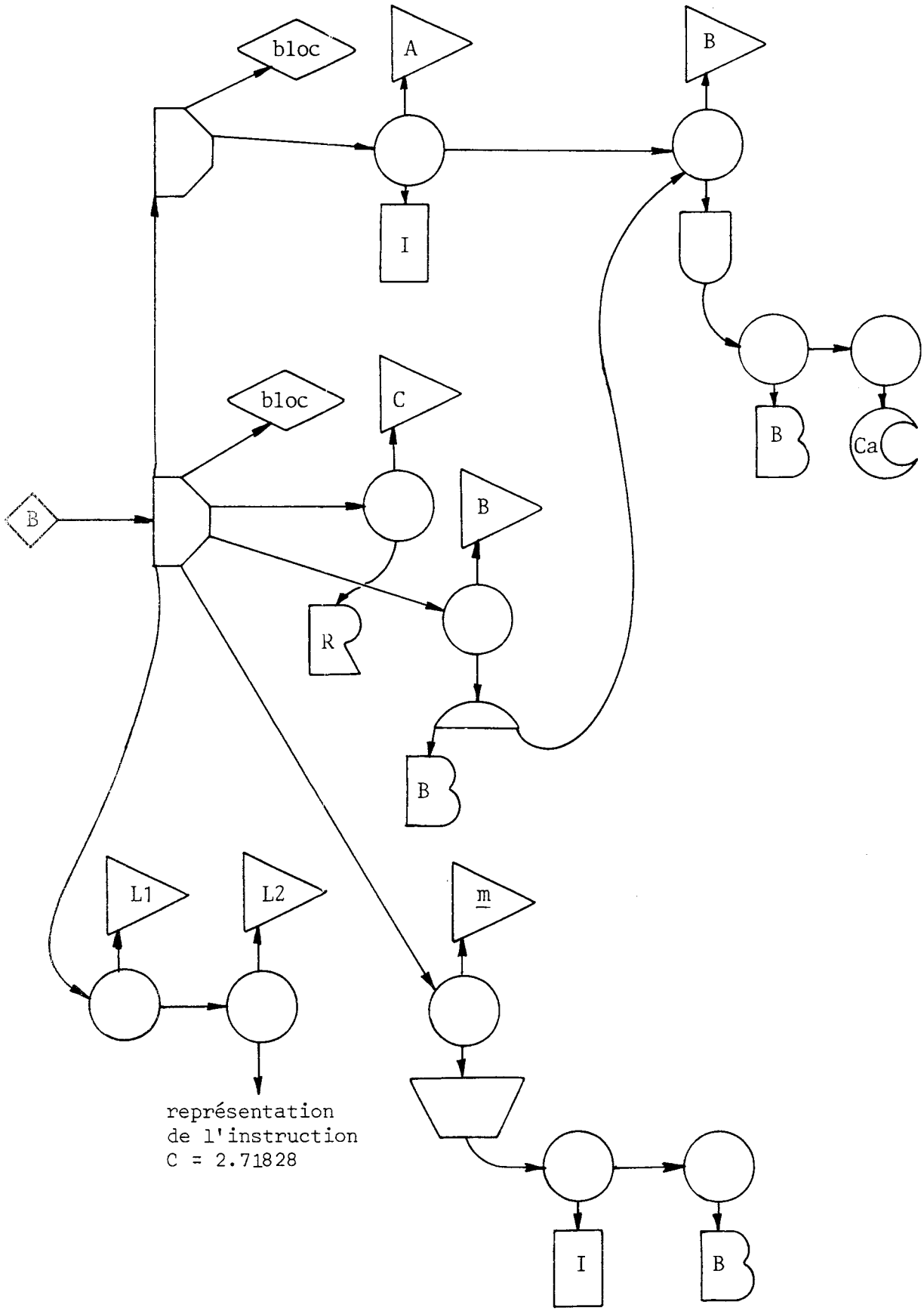
- F a deux rôles :
  - pendant le traitement de la partie déclaration, c'est la liste des déclarations additionnelles rencontrées ;
  - pendant le traitement de la partie instruction, c'est la liste des étiquettes définies ou utilisées.

On n'entrera pas ici dans le détail de la façon dont ces informations sont structurées. L'exemple suivant suffit à en donner une idée.

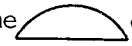
Soit le programme :

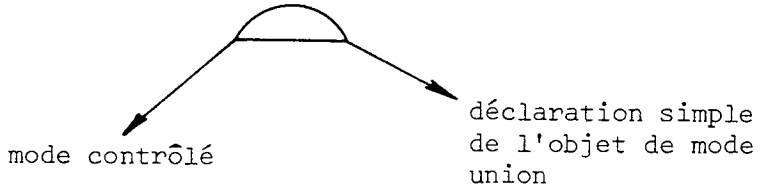
```
(A est ent, B est union (bool, car) dans A = 1 ; B = vrai ;
    (C est reel, B doitetre bool, m rep tuple (ent, bool) dans
        C = 3.14159 ; B = faux ; allera L1 ; L2 : C = 2.71828 ;
                                ...) ; ...)
```

Quand le traducteur vient de traiter l'instruction C = 2.71828, la pile des blocs a la configuration suivante (les arcs aboutissant à▼ ne sont pas représentés) :



représentation
de l'instruction
C = 2.71828

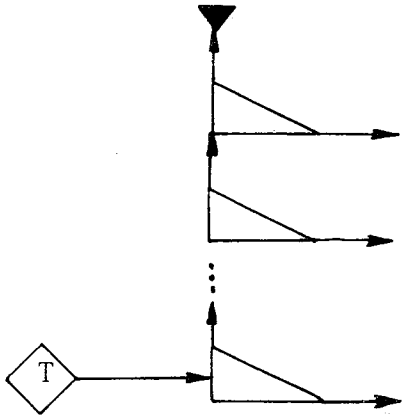
Remarque : la forme  est utilisée exclusivement dans la représentation d'une déclaration conditionnelle :



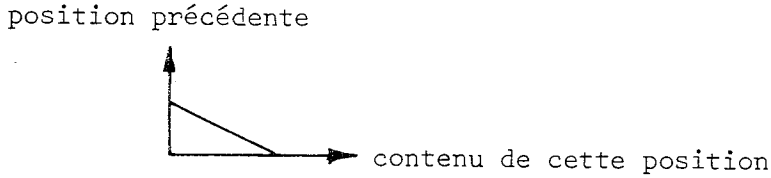
- Représentation abstraite d'un programme

Le sommet de la pile qui sert à construire la représentation abstraite d'un programme est indiqué par 

, et cette pile a la forme :

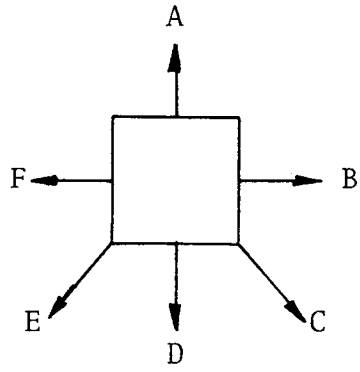


où chaque position de la pile est comprise comme :

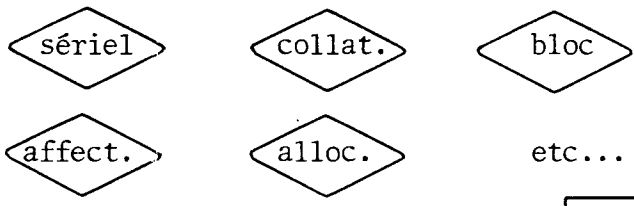


Cette pile est utilisée pour manipuler les objets et construire les structures de la sémantique statique (modes, déclarations, etc...), et en particulier pour former la structure et les instructions qui entrent dans la représentation abstraite d'un programme.

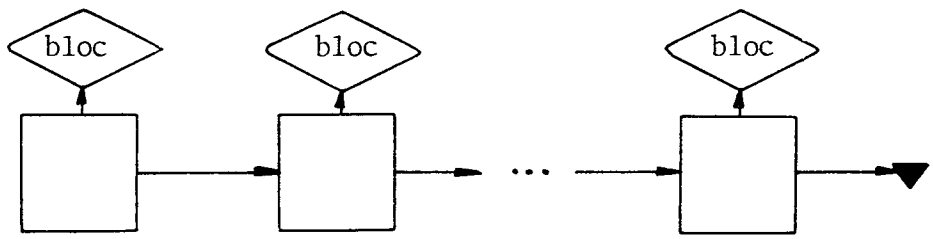
C'est un type d'objets unique qui sert à réunir toutes les informations sur la structure d'un programme, et en particulier à représenter les instructions :



- A indique la nature de l'information représentée par l'objet en question : A est un objet nommé, de type "losange". Exemples :



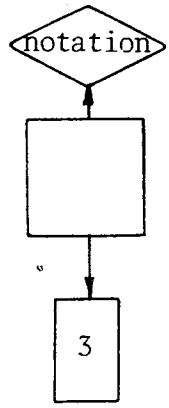
- B est l'élément suivant, s'il existe, de type  , dans la structure du programme abstrait. Par exemple, les blocs d'un corps d'expression composée sont chaînés entre eux de cette façon :



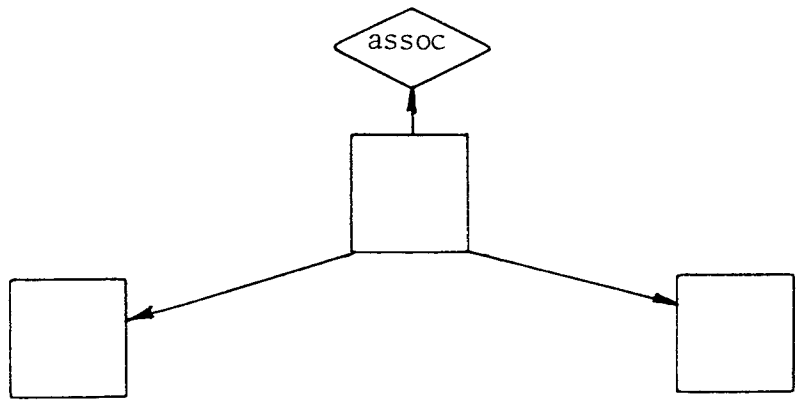
Il en est de même pour les branches d'instructions de la partie instruction d'un bloc, les séquences d'instructions d'une branche d'instructions et les instructions d'une séquence d'instructions.

- C, D et E sont les constituants de l'élément de programme représenté.

Leur nature dépend de l'objet considéré. Par exemple, pour une notation d'entier, considérée comme une expression, seul D sera utilisé :



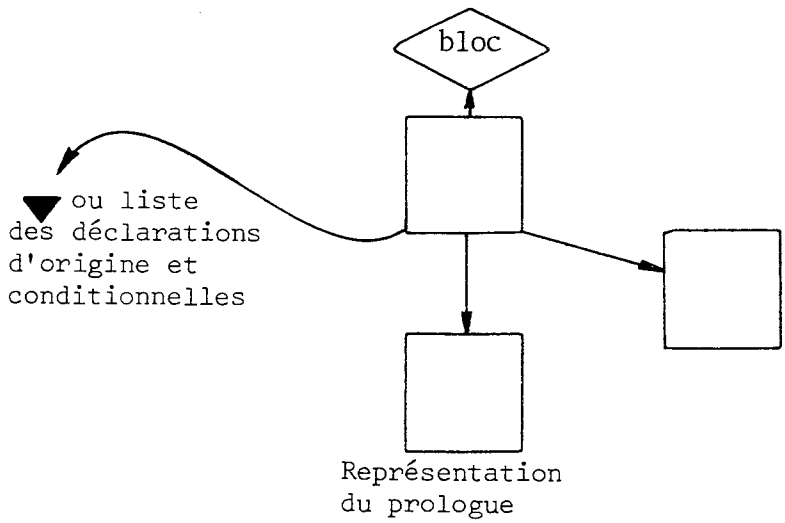
Autre exemple, pour une association seuls C et E sont utilisés :



Représentation de l'origine à identifier

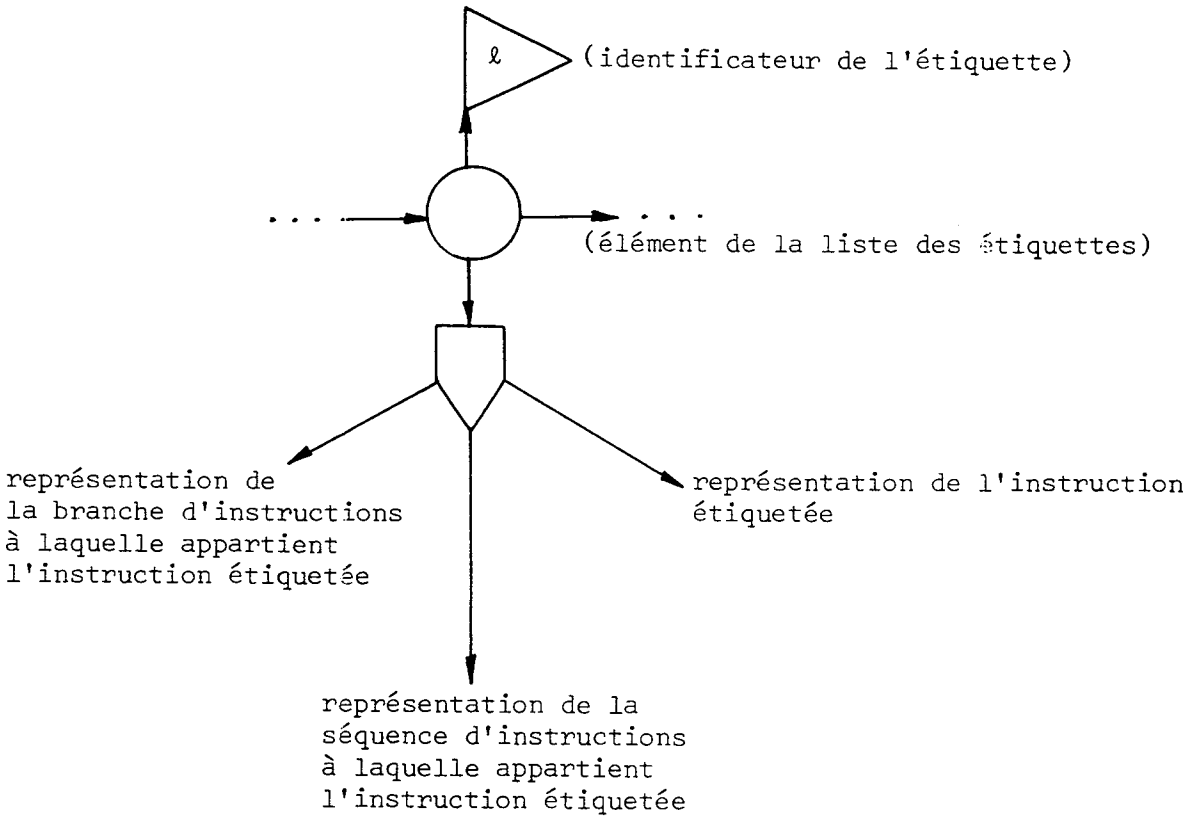
Représentation de l'expression, membre droit de l'association.

Pour un bloc, C, D et E sont utilisés tous les trois :



- Enfin, F, pendant la traduction, est le mode du résultat de l'instruction représentée et, pendant l'interprétation, il est ce résultat lui-même. C'est d'ailleurs là l'originalité principale dans le principe de fonctionnement de la définition formelle de BASEL.

De plus, à cause des problèmes de structure posés par les étiquettes et les instructions allera (malgré la sévère restriction sur leur emploi en BASEL), quand un bloc a été complètement traduit, les étiquettes qu'il contient sont toutes représentées par :



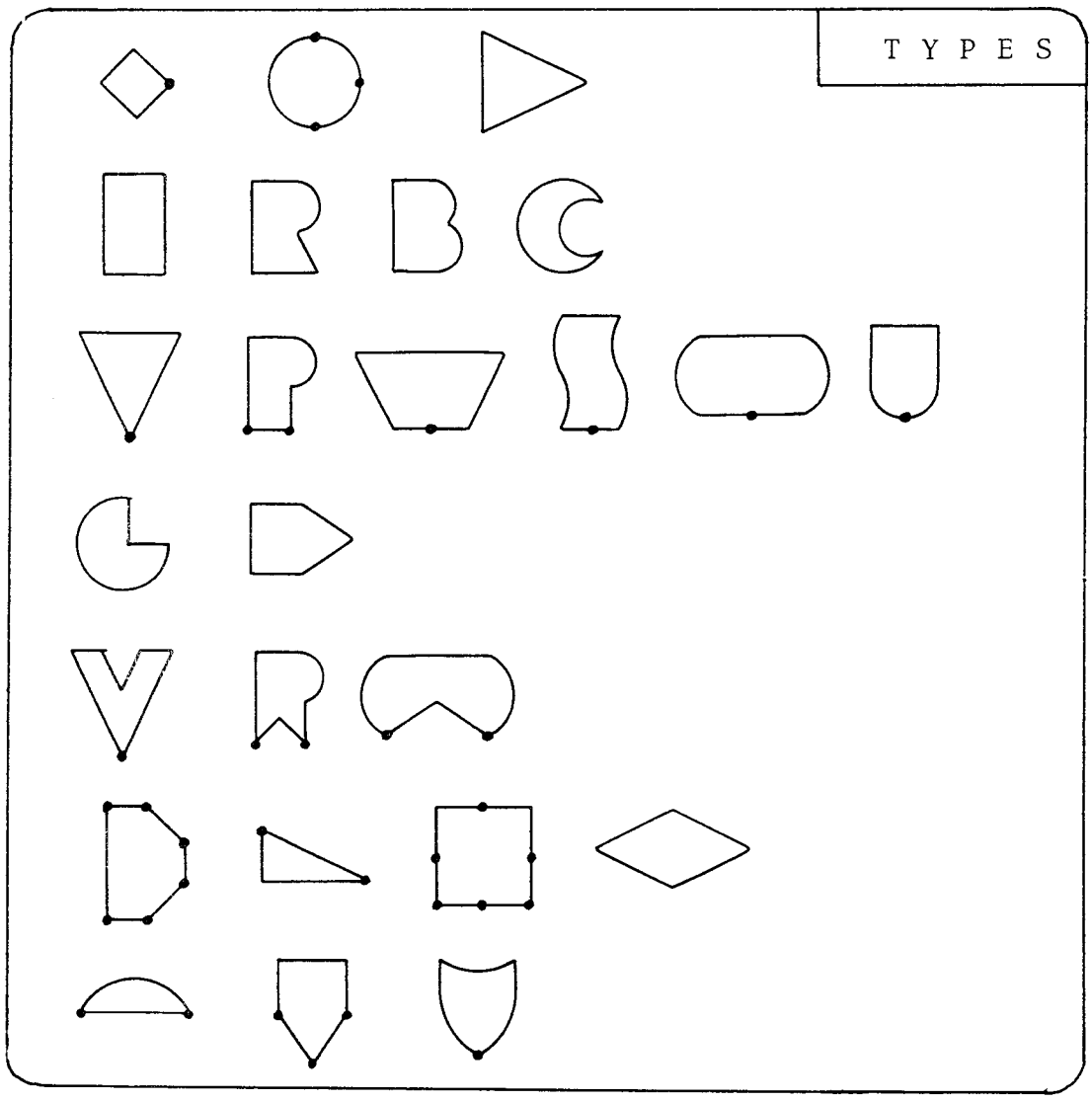
3.1.4. DEFINITION FORMELLE DE BASEL

Ce qui suit est une partie de la définition formelle de BASEL, construite selon les principes qui viennent d'être décrits. Quelques commentaires l'accompagnent.

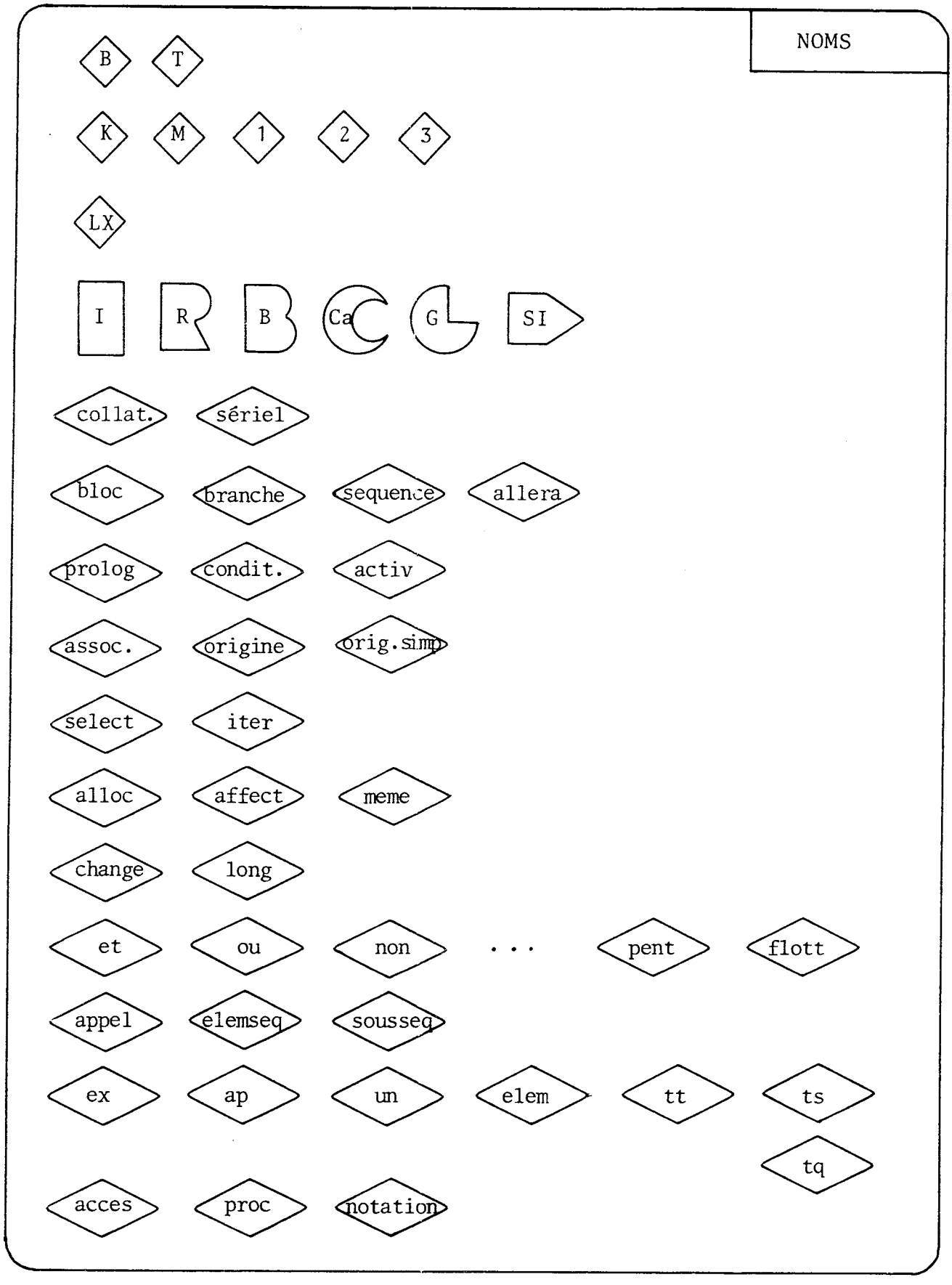
3.1.4.1. Description d'une partie du traducteur

a. Définition des types

D'après ce qui vient d'être dit, les divers types d'objets utilisés sont les suivants :



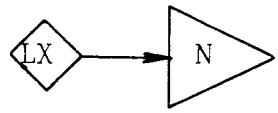
b. Objets nommés globaux



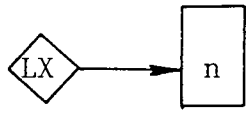
Les index ◇K◇ , ◇M◇ , ◇1◇ , ◇2◇ et ◇3◇ sont des index "de service" utilisés uniquement par le traducteur pour construire les éléments de la représentation abstraite : lorsqu'un objet de type □ est sur le point d'être construit, ce sont ces index qui indiquent les divers objets qui seront rattachés à cet élément de représentation abstraite.

◇LX◇ est un index qui donne accès à d'autres objets qui sont construits par une phase "LeXicographique" du traitement d'un programme pendant la traduction :

- quand un identificateur ou un symbole N est rencontré pendant la lecture du programme concret, l'objet correspondant (un "nom") est indiqué par ◇LX◇ :



- quand une notation d'entier n est rencontrée pendant la lecture du programme concret, l'objet nommé correspondant est indiqué par ◇LX◇ :



- même chose pour les notations de réels, de booléens et de caractères.

Les objets nommés globaux obtenus par utilisation de ◇LX◇ sont donc en nombre potentiellement infini, et ne sont donc pas mentionnés dans la définition des objets nommés globaux.

Enfin le "type d'instruction" ◇elem◇ correspond à une opération qui n'existe pas en BASEL "concret" mais qui est nécessaire pour construire la représentation abstraite d'une modification composée ; en effet, il faut pourvoir extraire un élément d'un tuple afin de lui faire subir des modifications, et une telle opération n'existe pas en BASEL : c'est le rôle de l'opération indiquée par le "losange" ◇elem◇ .

c. Syntaxe et appels

Je donne ici une liste presque complète (toutes les opérations n'y sont pas) des règles de grammaire de BASEL telles qu'elles sont utilisées par le traducteur. Elles sont accompagnées des appels des procédures de la sémantique statique.

```
programme ::= expression-composée
                    PROGRAMME
expression-composée ::= expression-composée-serielle
expression-composée ::= expression-composée-collateralle
expression-composée-serielle ::= debut-seriel corps-d-expression-composée
                                      fin-seriel
                    COMPOSEE SERIELLE
debut-seriel ::= (
                    OUVRIR BLOC
fin-seriel ::= )
                    FERMER BLOC
expression-composée-collateralle ::= debut-collateral
                                      corps-d-expression-composée fin-collatera
                    COMPOSEE COLLATERALLE
debut-collateral ::= [
                    OUVRIR BLOC
fin-collateral ::= ]
                    FERMER BLOC
corps-d-expression-composée ::= corps-d-expression-composée fin-debut  bloc
                    CHAINER BLOC
corps-d-expression-composée ::= bloc
fin-debut ::= ,
                    FERMER ET OUVRIR BLOC
bloc ::= partie-declaration dans partie-instruction
                    BLOC
bloc ::= partie-instruction
                    BLOC
partie-declaration ::= declarations
                    DECLARATIONS
declarations ::= declarations , declaration
```

```
declarations ::= declaration
declaration ::= declaration-de-mode
declaration ::= declaration-de-generique
declaration ::= declaration-simple
declaration ::= declaration-additionnelle
declaration ::= declaration-conditionnelle
declaration-de-mode ::= symbole rep mode
                        DECLARER MODE
declaration-de-generique ::= identificateur est générique
                        DECLARER GENERIQUE
declaration-simple ::= identificateur est mode
                        DECLARER ORIGINE
declaration-additionnelle ::= identificateur estaussi mode
                        DECLARER SIGNIFICATION
declaration-conditionnelle ::= identificateur doitêtre mode
                        DECLARER CONDITION

mode ::= mode-entier
mode ::= mode-reel
mode ::= mode-booleen
mode ::= mode-caractère
mode ::= mode-variable
mode ::= mode-procédure
mode ::= mode-tuple
mode ::= mode-structure
mode ::= mode-séquence
mode ::= mode-union
mode ::= symbole
mode-entier ::= ent
                        MODE ENTIER
mode-reel ::= reel
                        MODE REEL
mode-booleen ::= bool
                        MODE BOOLEEN
mode-caractère ::= car
                        MODE CARACTERE
mode-variable ::= var mode
                        MODE VARIABLE
```

```
mode-procedure ::= proc modes resultat
                        MODE PROCEDURE AVEC PARAMETRES
mode-procedure ::= proc resultat
                        MODE PROCEDURE SANS PARAMETRE
resultat ::= mode
resultat ::= rien
                        RIEN
modes ::= (liste-de-modes)
                        MODES
liste-de-modes ::= liste-de-modes , element-de-liste-de-modes
                        CHAINER LISTE MODES
liste-de-mode ::= element-de-liste-de-modes
element-de-liste-de-modes ::= mode
                        ELEMENT DE LISTE DE MODES
mode-tuple ::= tuple modes
                        MODE TUPLE
mode-structure ::= struct champs
                        MODE STRUCTURE
champs ::= (liste-de-champs)
                        CHAMPS
liste-de-champs ::= liste-de-champs , champ
                        CHAINER LISTE DE CHAMPS
liste-de-champs ::= champ
champ ::= mode identificateur
                        CHAMP
mode-sequence ::= seq mode
                        MODE SEQUENCE
mode-union ::= union modes
                        MODE UNION
partie-instruction ::= partie-instruction | branche-d-instructions
                        CHAINER BRANCHES
partie-instruction ::= branche-d-instructions
branche-d-instructions ::= suite-de-conditions
                        BRANCHE D'INSTRUCTIONS
suite-de-conditions ::= suite-de-conditions → sequence-d-instructions
                        CHAINER SEQUENCES
suite-de-conditions ::= sequence-d-instructions
```

```
sequence-d-instructions ::= suite-d-instructions
                        SEQUENCE D'INSTRUCTIONS
suite-d-instructions ::= suite-d-instructions ; instruction
                        CHAINER INSTRUCTIONS
suite-d-instructions ::= instruction
instruction ::= identificateur : instructions
                        DECLARER ETIQUETTE
instruction ::= allera
instruction ::= expression
etiquette ::= identificateur :
allera ::= allera identificateur
                        ALLER A
expression ::= origine = expression
                        ASSOCIATION
expression ::= terme
origine ::= identificateur de origine
                        ORIGINE
origine ::= origine-simple
origine-simple ::= identificateur
                        ORIGINE SIMPLE
terme ::= identificateur de terme
                        SELECTION
terme ::= facteur
facteur ::= primaire fois facteur
                        ITERATION
facteur ::= primaire
primaire ::= opération
primaire ::= opérande
opération ::= alloc operande
                        ALLOCATION
opération ::= affect operande
                        AFFECTATION
opération ::= même opérande
                        IDENTITE
...
opération ::= flott opérande
```

```
opérande ::= opérande element
                    APPLICATION
opérande ::= élément
element ::= mode-cible element
                    MODIFICATION
element ::= base
mode-cible ::= mode
                    MODE
base ::= accès
base ::= texte-de-procédure
base ::= notation
base ::= expression-composée
accès ::= identificateur
                    ACCES
texte-de-procédure ::= debut-procédure partie-parametre dans
                                        partie-instruction fin-procedure
                    PROCEDURE
texte-de-procédure ::= debut-procedure partie-instruction fin-procédure
                    PROCEDURE
debut-procédure ::= <
                    OUVRIR PROCEDURE
fin-procédure ::= >
                    FERMER PROCEDURE
partie-parametre ::= parametres
                    PARAMETRES
parametres ::= parametres , declaration-simple
parametres ::= declaration-simple
notation ::= entier
                    ENTIER
notation ::= reel
                    REEL
notation ::= booleen
                    BOOLEEN
notation ::= caractere
                    CARACTERE
caractère ::= 'marque'
```

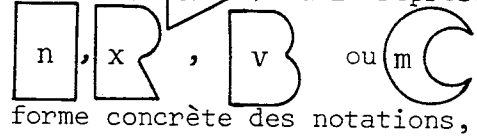
```
notation ::= nul mode-de-valeur-nulle
                    VALEUR NULLE
mode-de-valeur-nulle ::= mode
                    MODE
notation ::= vide
                    VIDE
notation ::= chaîne
                    CHAINE
chaîne ::= "suite-d-elements-de-chaîne"
suite-d-elements-de-chaîne ::= suite-d-elements-de-chaîne element-de-chaine
suite-d-elements-de-chaîne ::= element-de-chaîne
element-de-chaîne ::= marque
                    ELEMENT DE CHAINE
```

- Règles de niveau lexicographique :

```
identificateur ::= id
                    IDENTIFICATEUR
symbole ::= sy
                    SYMBOLE
entier ::= en
reel ::= re
booleen ::= bo
marque ::= ma
```

Ce sont ces règles qui, par un mécanisme qu'il n'est pas nécessaire de préciser, fait indiquer par 

un nom (

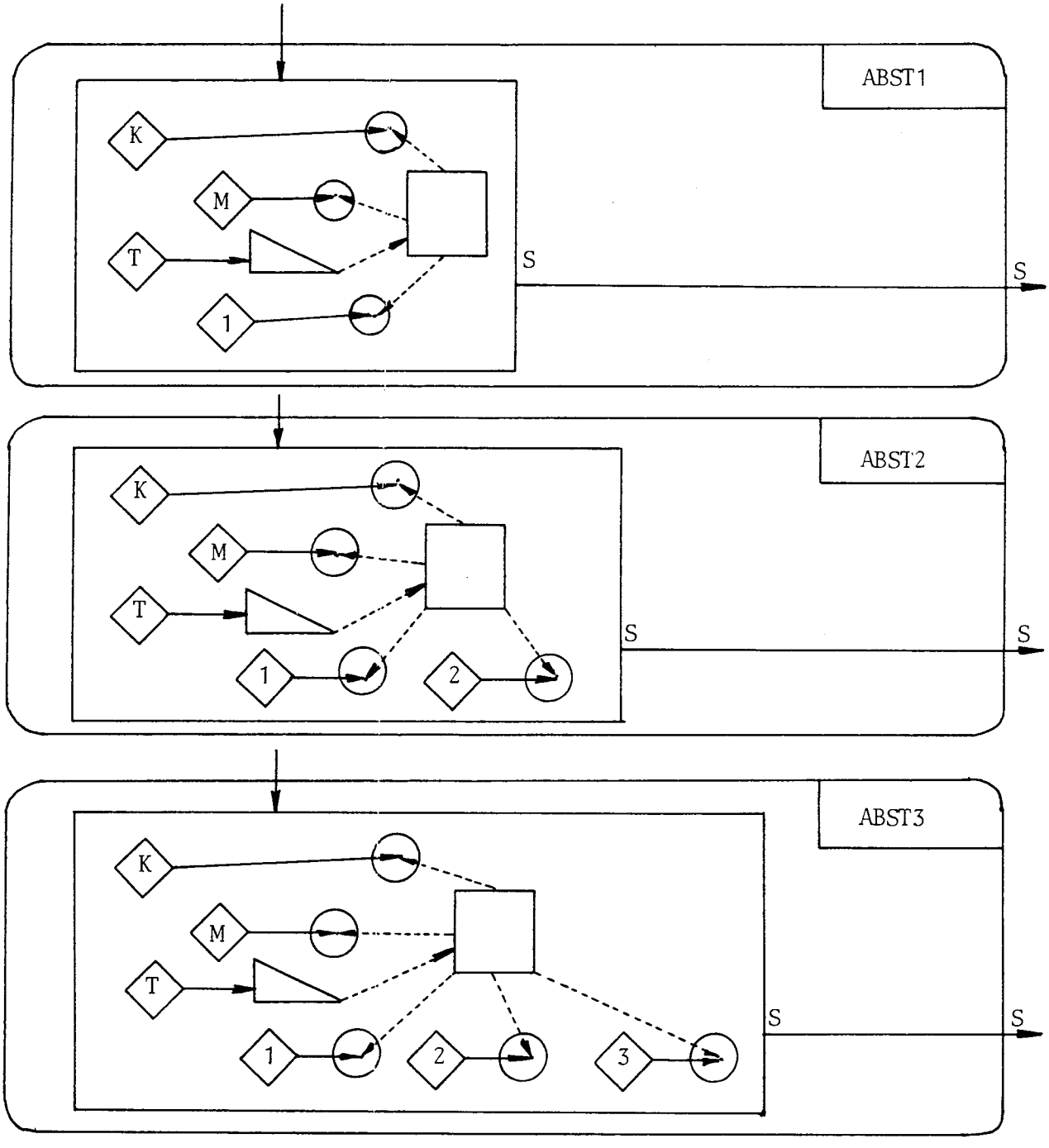
) ou la représentation d'une valeur d'un mode de base (

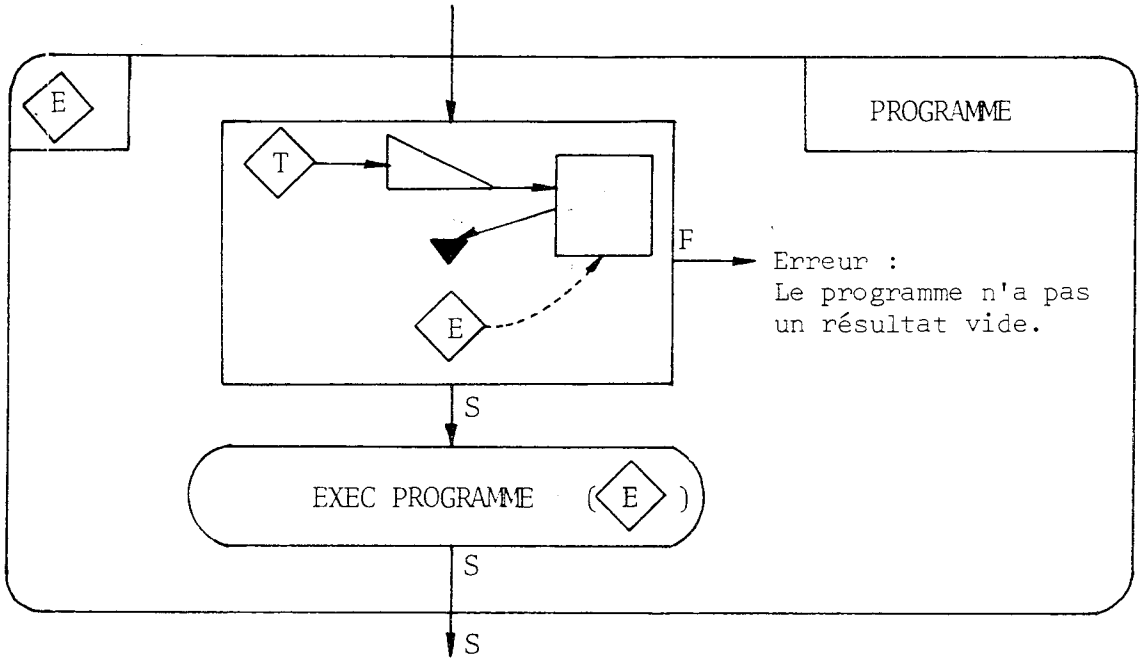
). (Il n'est pas utile de répéter ici la forme concrète des notations, déjà donnée plus haut dans la description de BASEL).

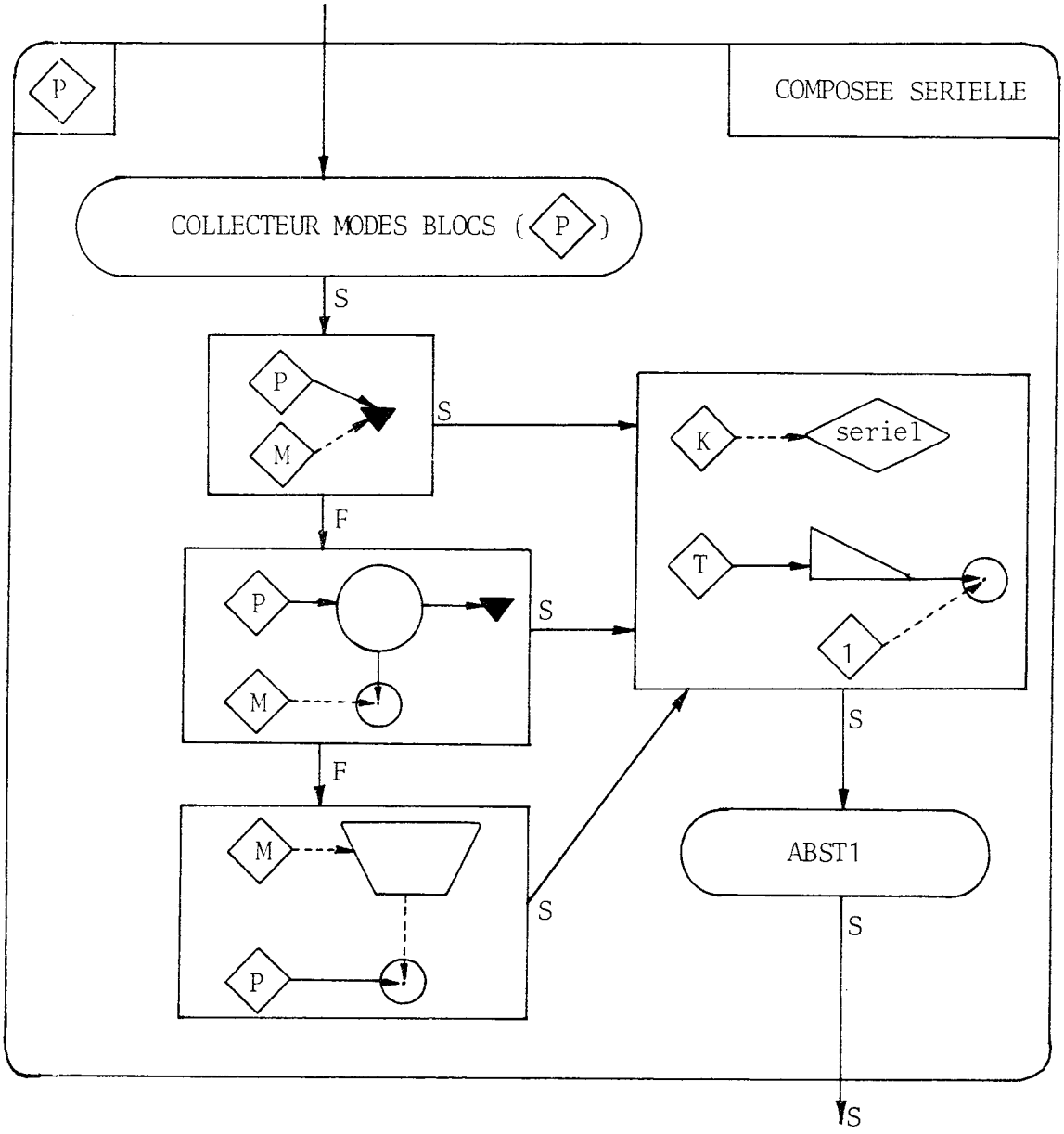
d. Procédures de la sémantique statique

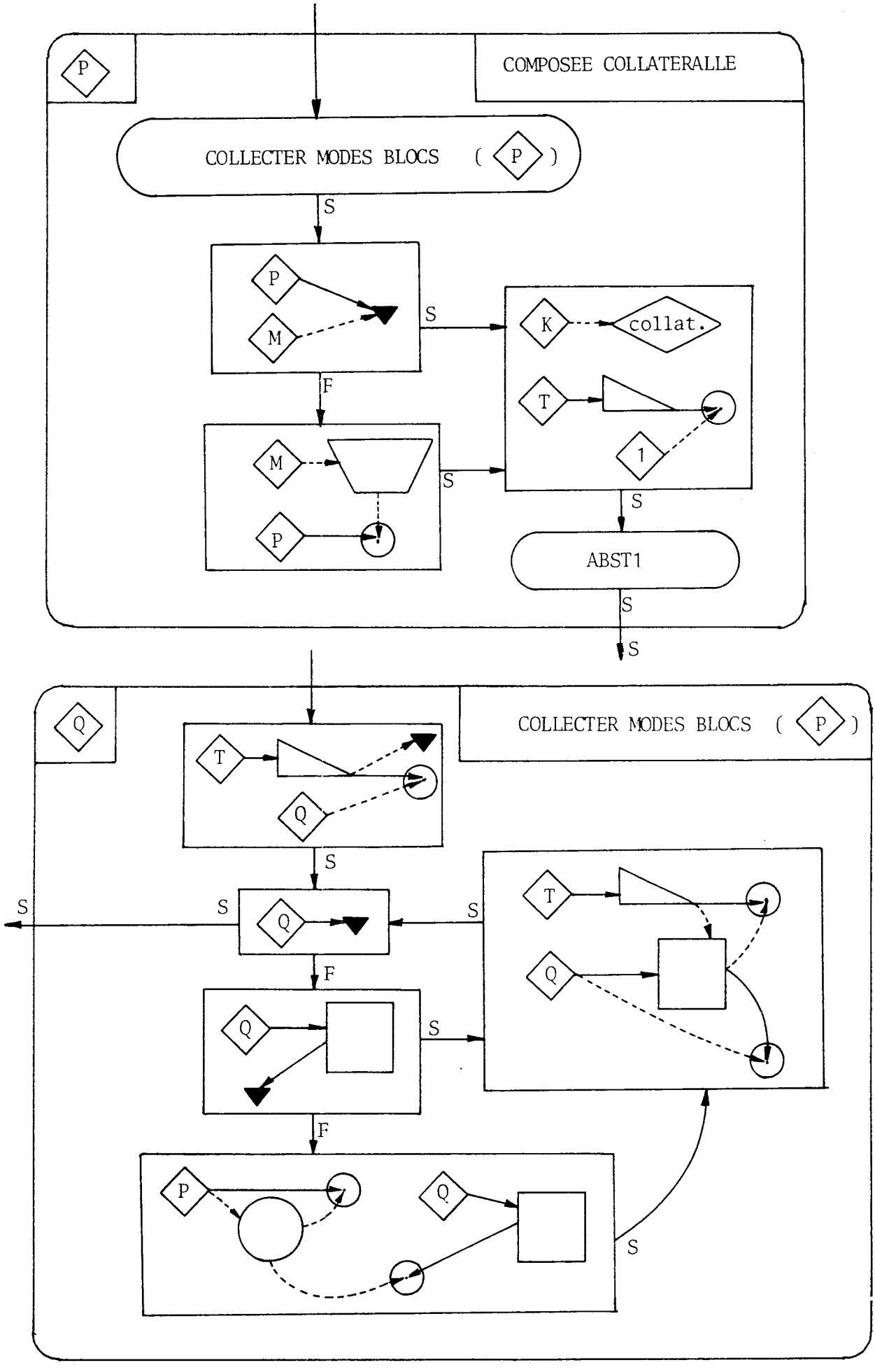
Je ne donne pas ici toutes les procédures de la sémantique statique, mais seulement certaines d'entre elles pour montrer de quelle façon elles opérent en général.

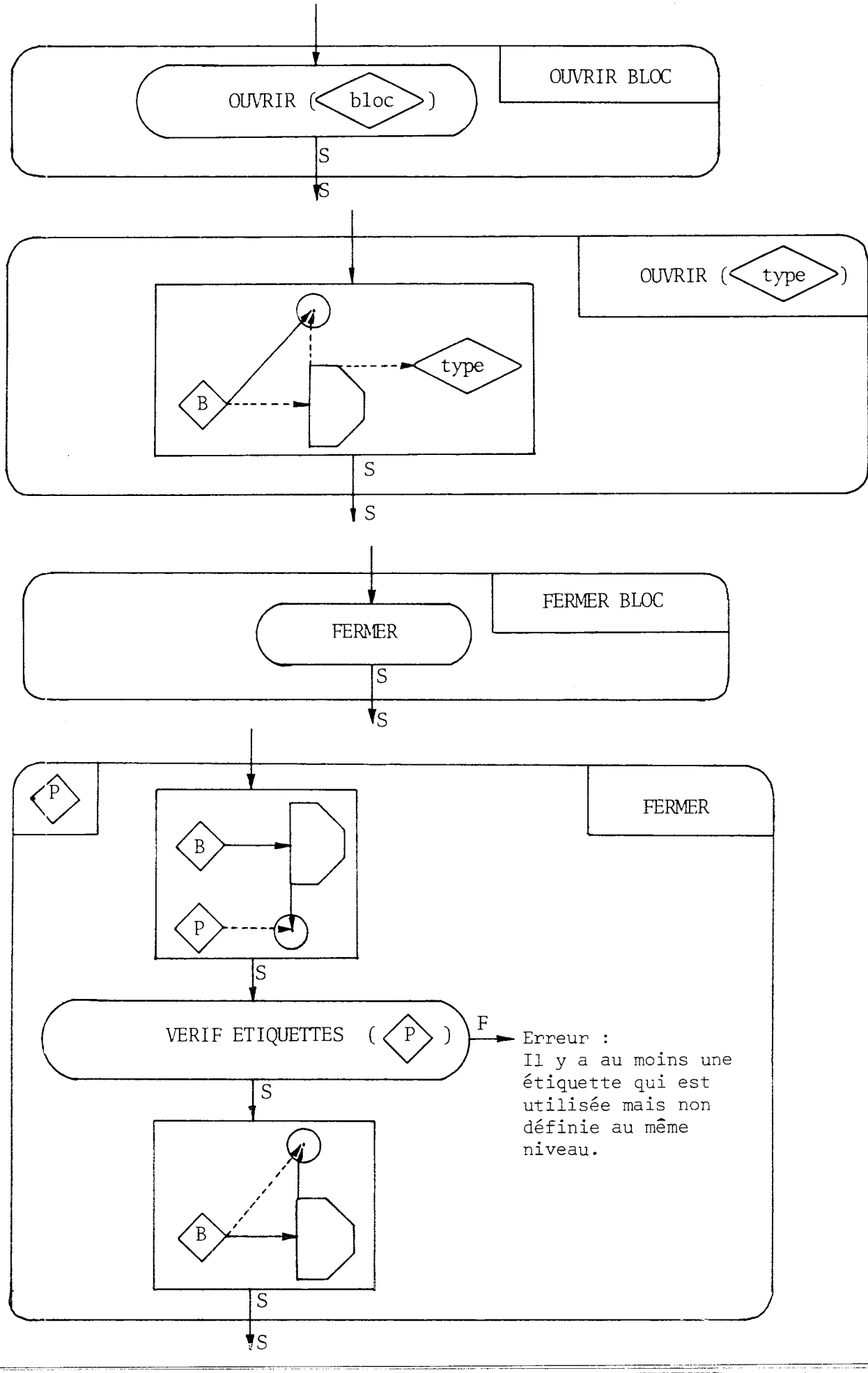
Dans les pages qui suivent, ces procédures sont rangées dans l'ordre où leurs appels sont écrits dans la syntaxe donnée précédemment, avec, assez souvent, des procédures de service intercallées. D'autre part, trois procédures de service sont utilisées fréquemment. Ce sont celles qui servent à construire la structure et les instructions du programme abstrait :

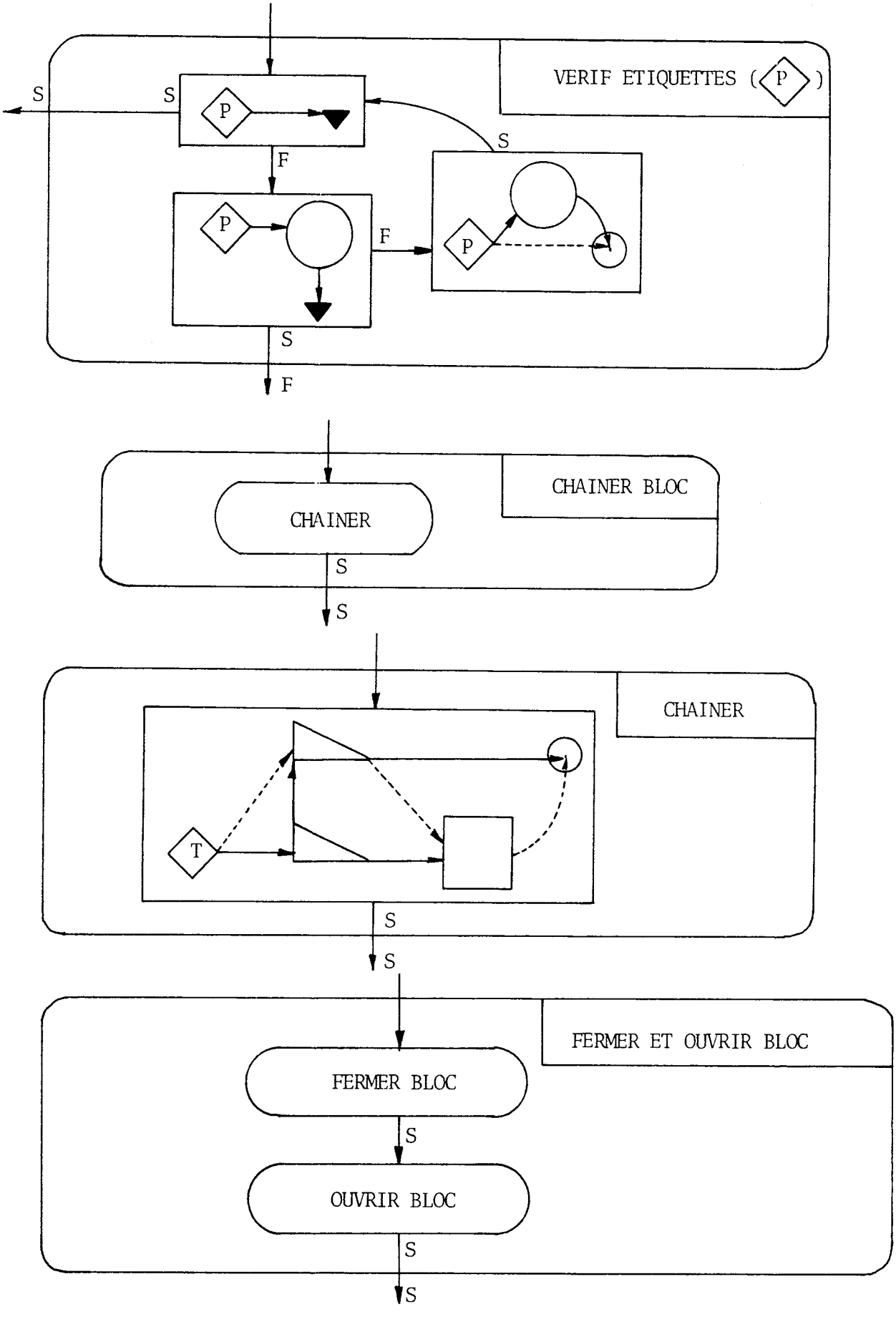


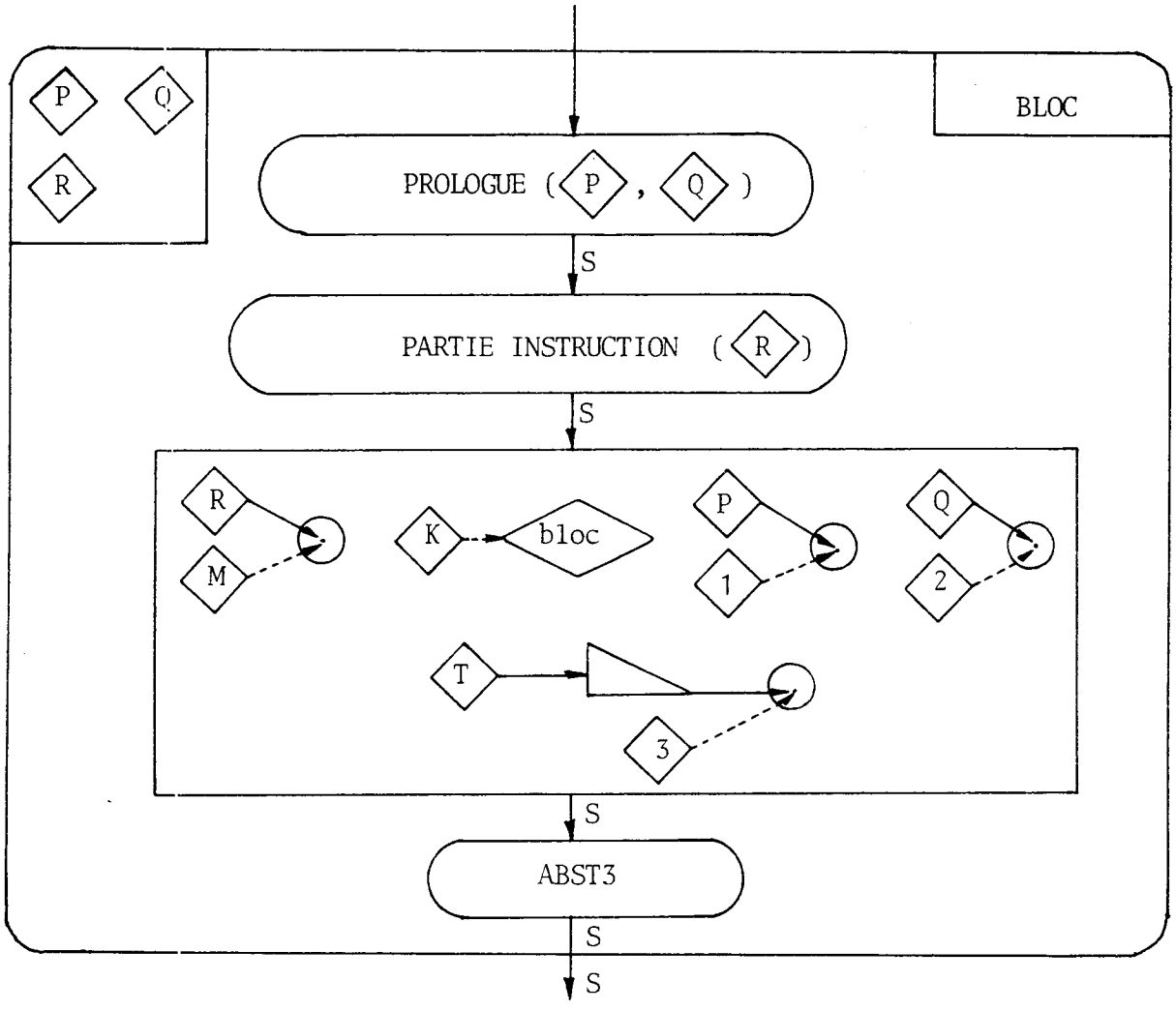

E
PROGRAMME
T
F
Erreur :
Le programme n'a pas
un résultat vide.
E
S
EXEC PROGRAMME (E)
S
S

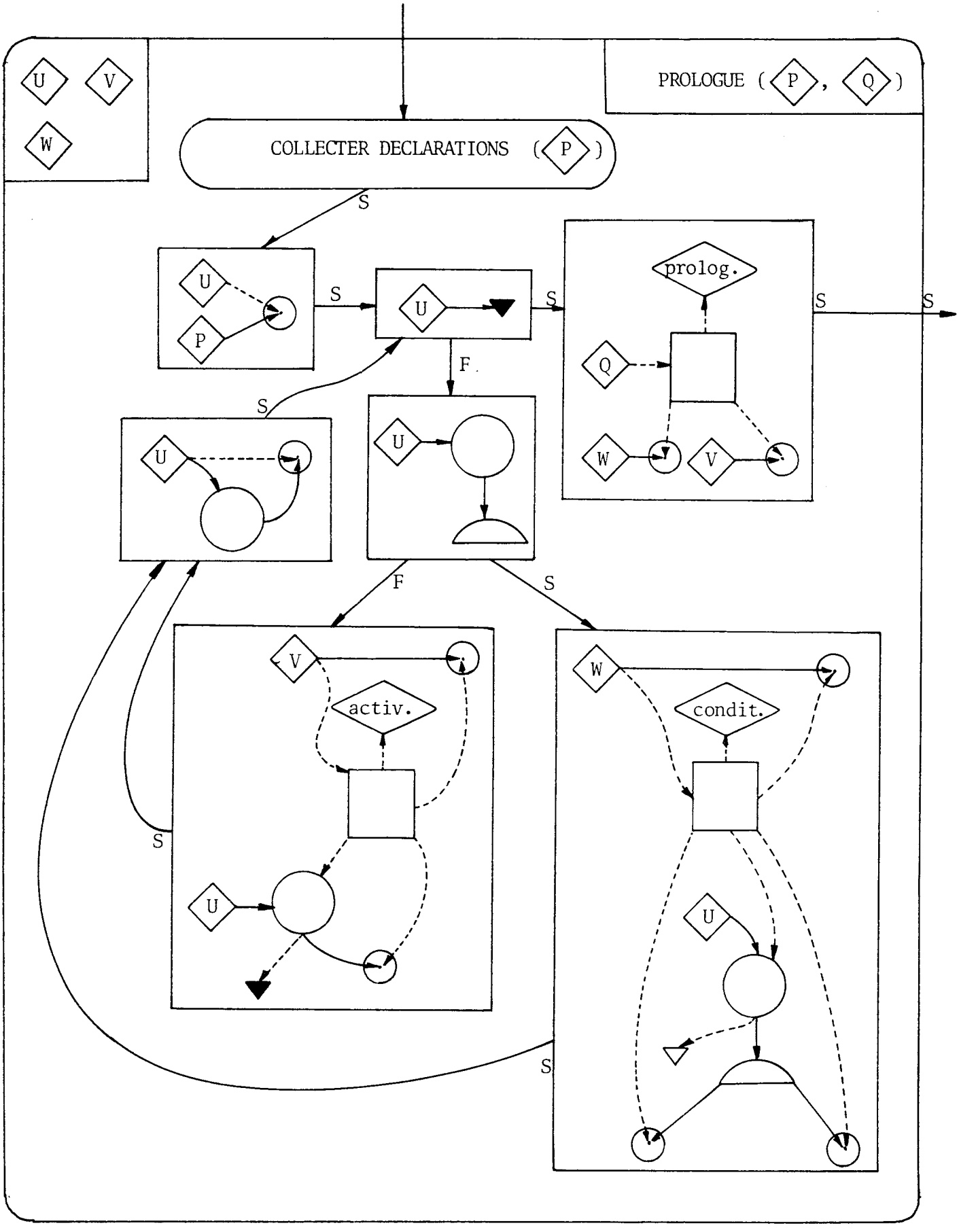

P
COMPOSEE SERIELLE
COLLECTEUR MODES BLOCS (P)
S
P
M
S
K
seriel
F
T
P
S
1
M
S
F
M
S
ABST1
P
S
S

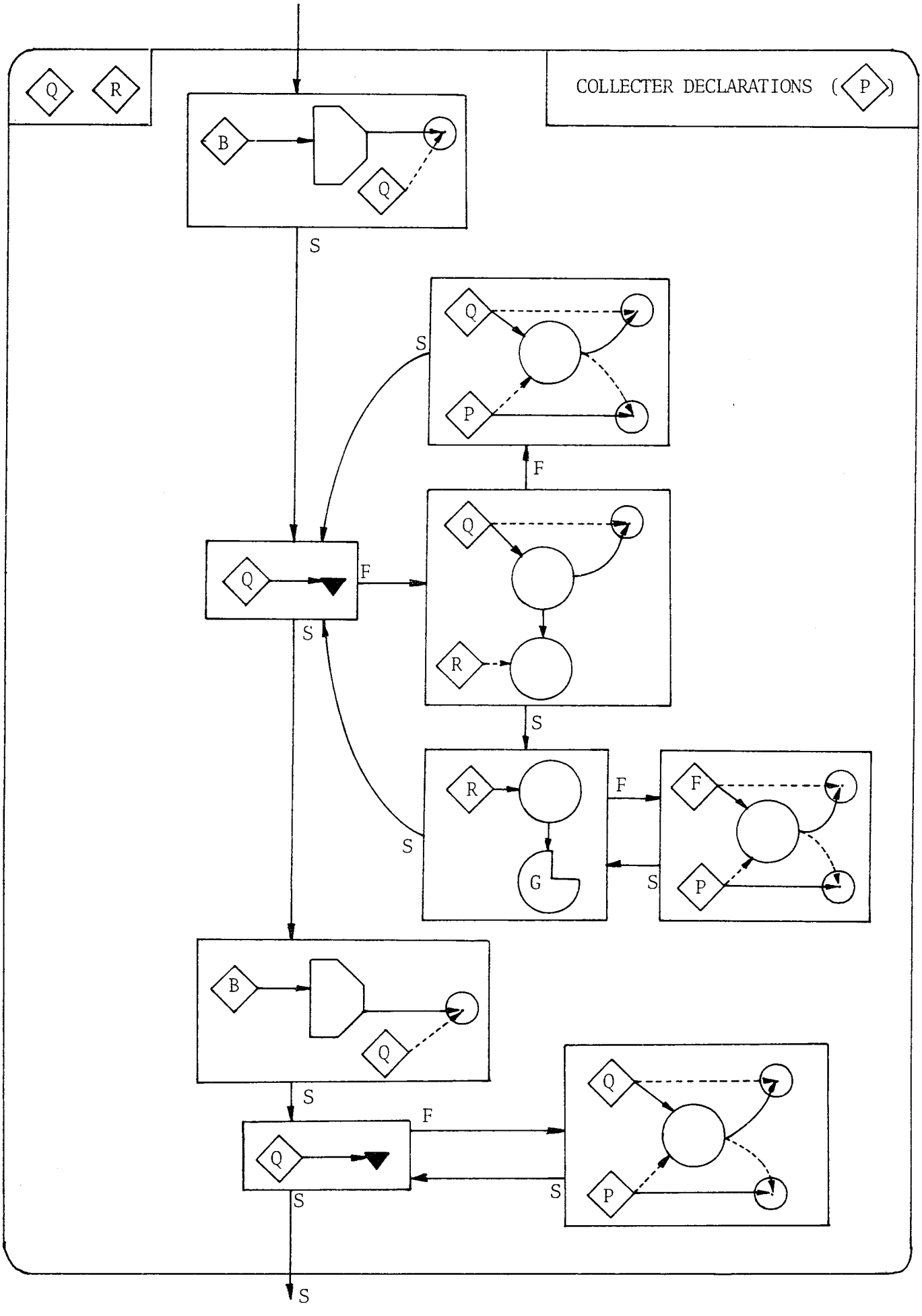

P
COMPOSEE COLLATERALLE
COLLECTER MODES BLOCS (P)
S
P
M
S
K
collat.
F
T
1
M
S
P
S
ABST1
S
S
Q
COLLECTER MODES BLOCS (P)
T
Q
S
S
S
S
Q
T
F
Q
Q
S
F
P
Q
S

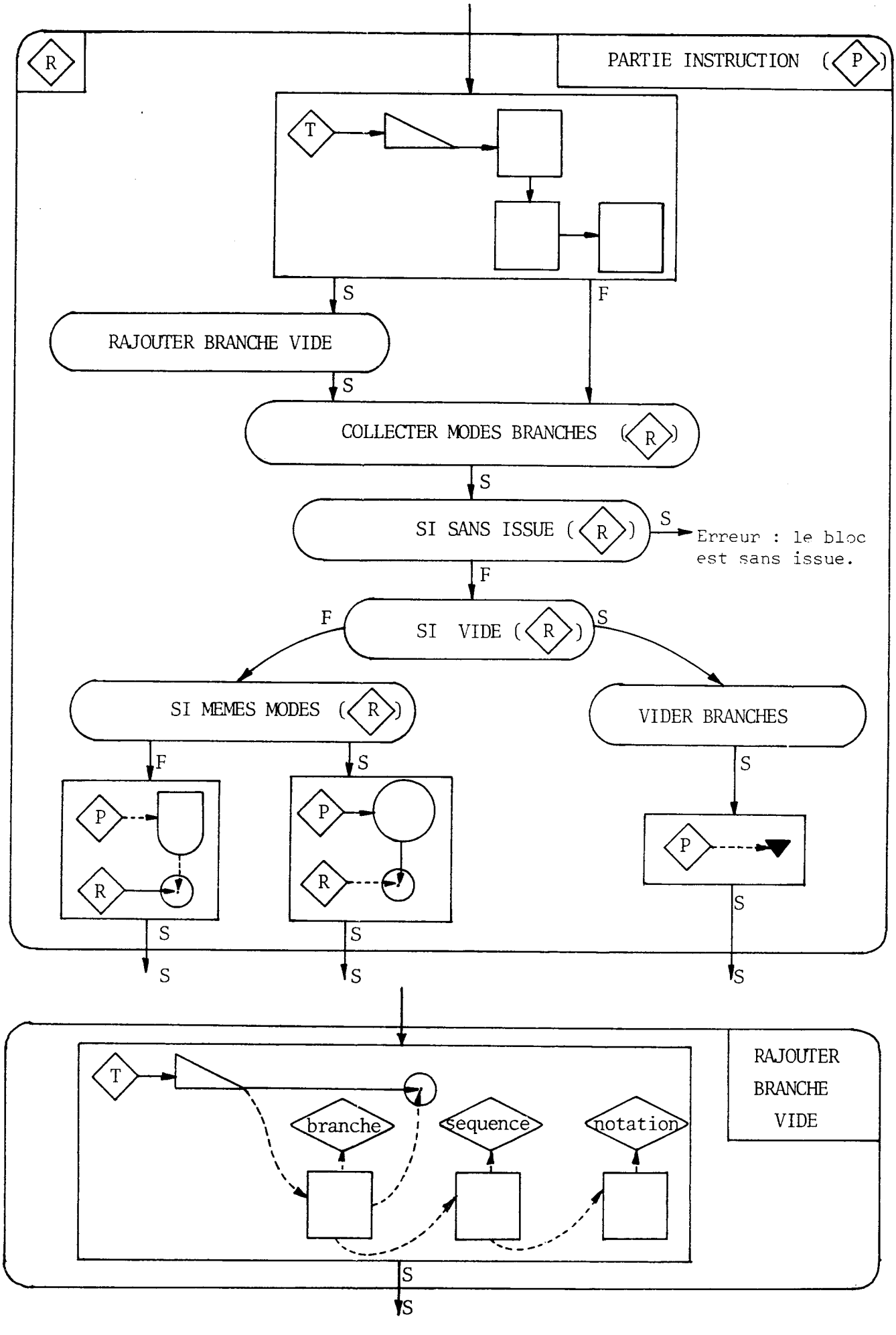

OUVRIR BLOC
OUVRIR (bloc)
S
S
OUVRIR (type)
type
B
S
S
FERMER BLOC
FERMER
S
S
P
FERMER
B
P
S
VERIF ETIQUETTES (P)
F
Erreur :
Il y a au moins une étiquette qui est utilisée mais non définie au même niveau.
S
B
S
S

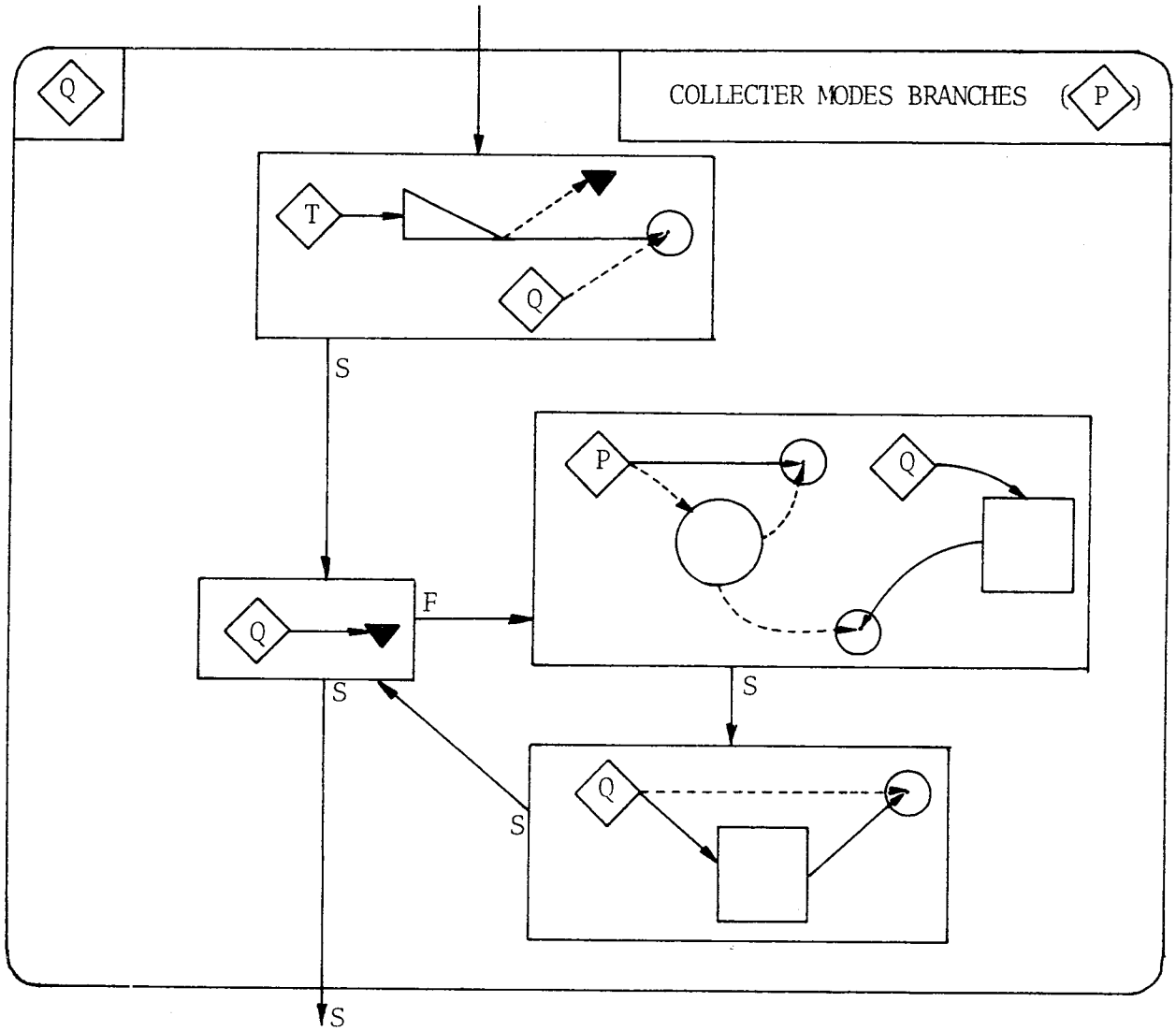

VERIF ETIQUETTES (P)
P
S
F
CHAINER BLOC
CHAINER
CHAINER
T
FERMER ET OUVRIR BLOC
FERMER BLOC
OUVRIR BLOC

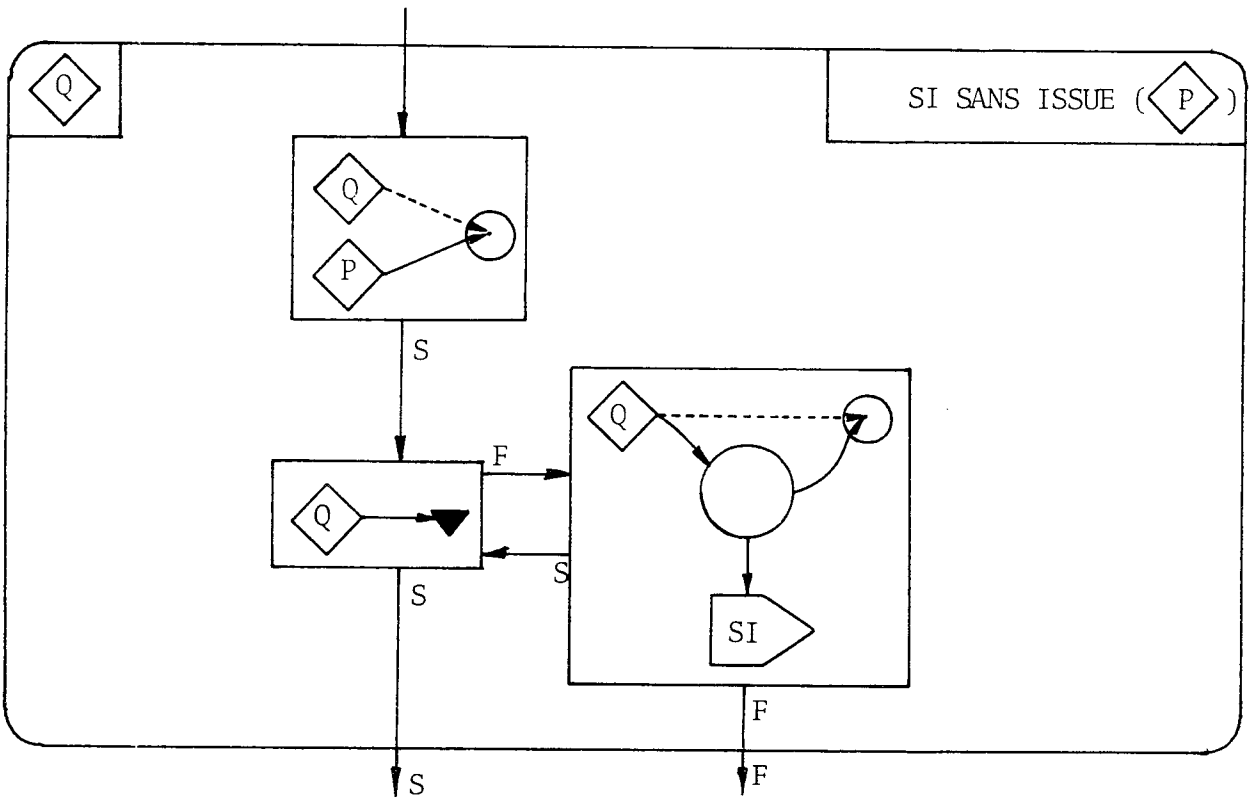

P
Q
R
BLOC
PROLOGUE (P , Q)
S
PARTIE INSTRUCTION (R)
S
R
M
K
bloc
P
1
Q
2
T
3
S
ABST3
S
S

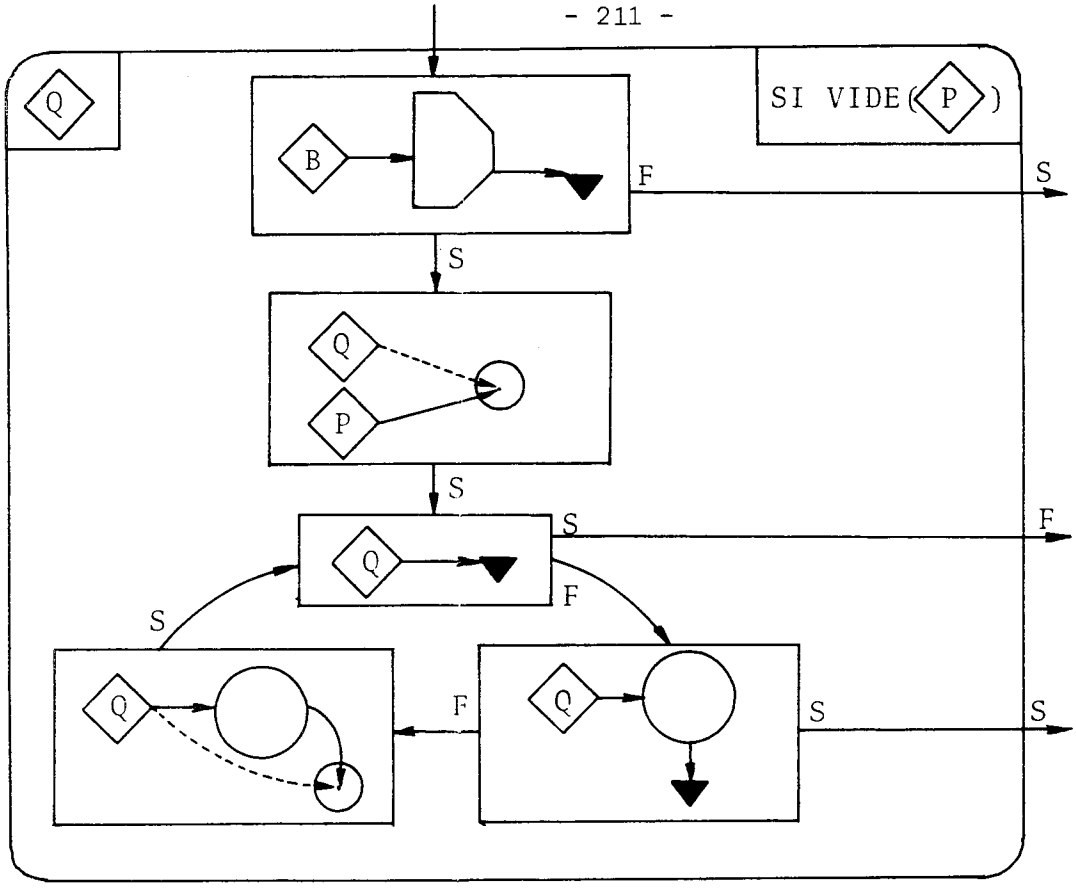

U
V
W
PROLOGUE (P , Q)
COLLECTER DECLARATIONS (P)
S
U
P
S
U
S
prolog.
Q
W
V
S
S
F
S
U
U
F
S
V
activ.
S
U
W
condit.
U
S

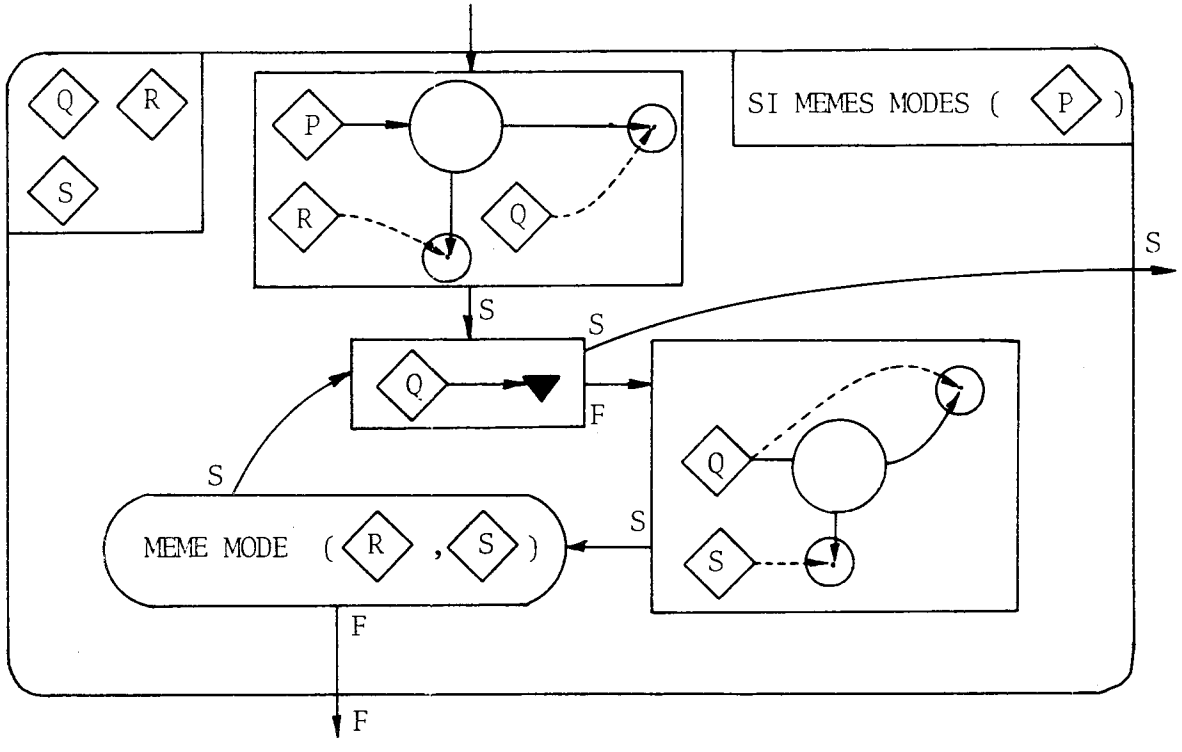

Q
R
COLLECTER DECLARATIONS (P)
B
Q
S
Q
S
P
F
Q
F
S
Q
R
S
R
F
F
S
G
S
P
B
Q
S
F
Q
S
S
Q
P
S


R
PARTIE INSTRUCTION (P)
T
S
F
RAJOUTER BRANCHE VIDE
S
COLLECTER MODES BRANCHES (R)
S
SI SANS ISSUE (R)
S
Erreur : le bloc est sans issue.
F
F
SI VIDE (R)
S
SI MEMES MODES (R)
VIDER BRANCHES
F
S
S
P
R
P
R
P
S
S
S
S
S
S
RAJOUTER BRANCHE VIDE
T
branche
sequence
notation
S
S

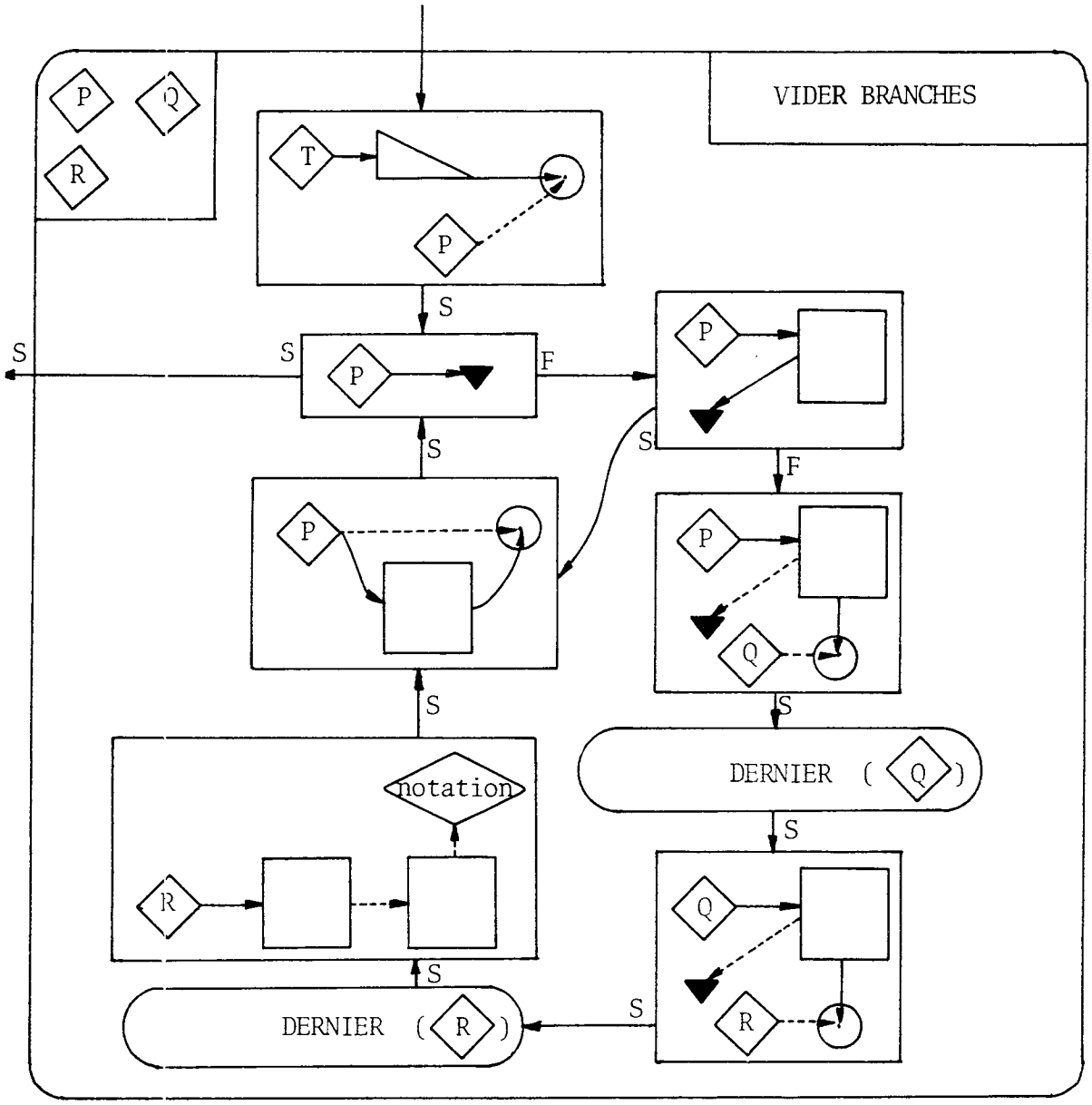

Q
COLLECTER MODES BRANCHES (P)
T
Q
S
P
Q
Q
F
S
S
Q
S
S

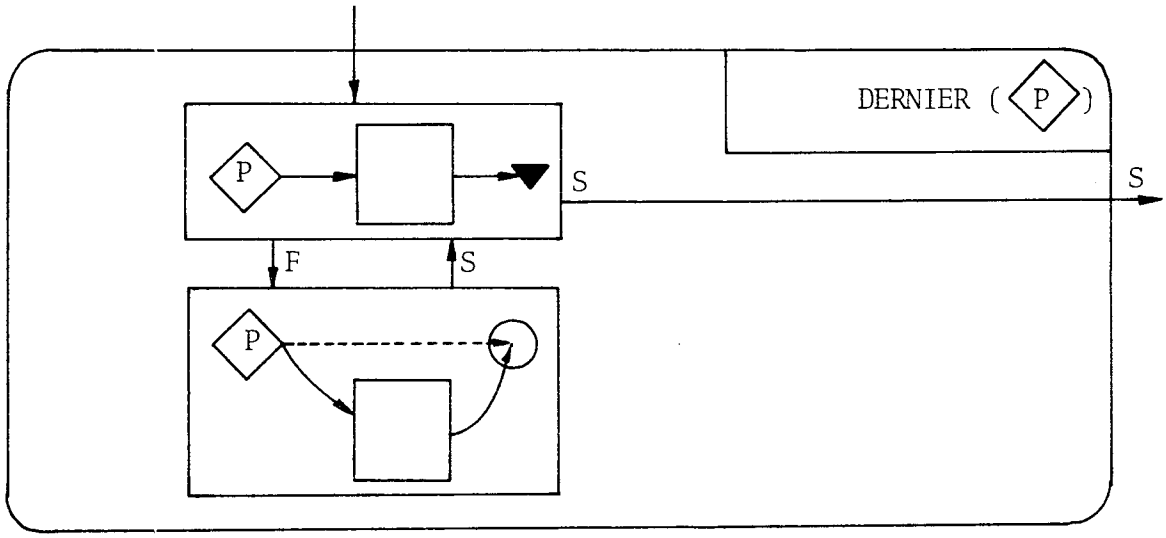

Q
SI SANS ISSUE (P)
Q
P
S
Q
F
Q
S
S
SI
F
S
F

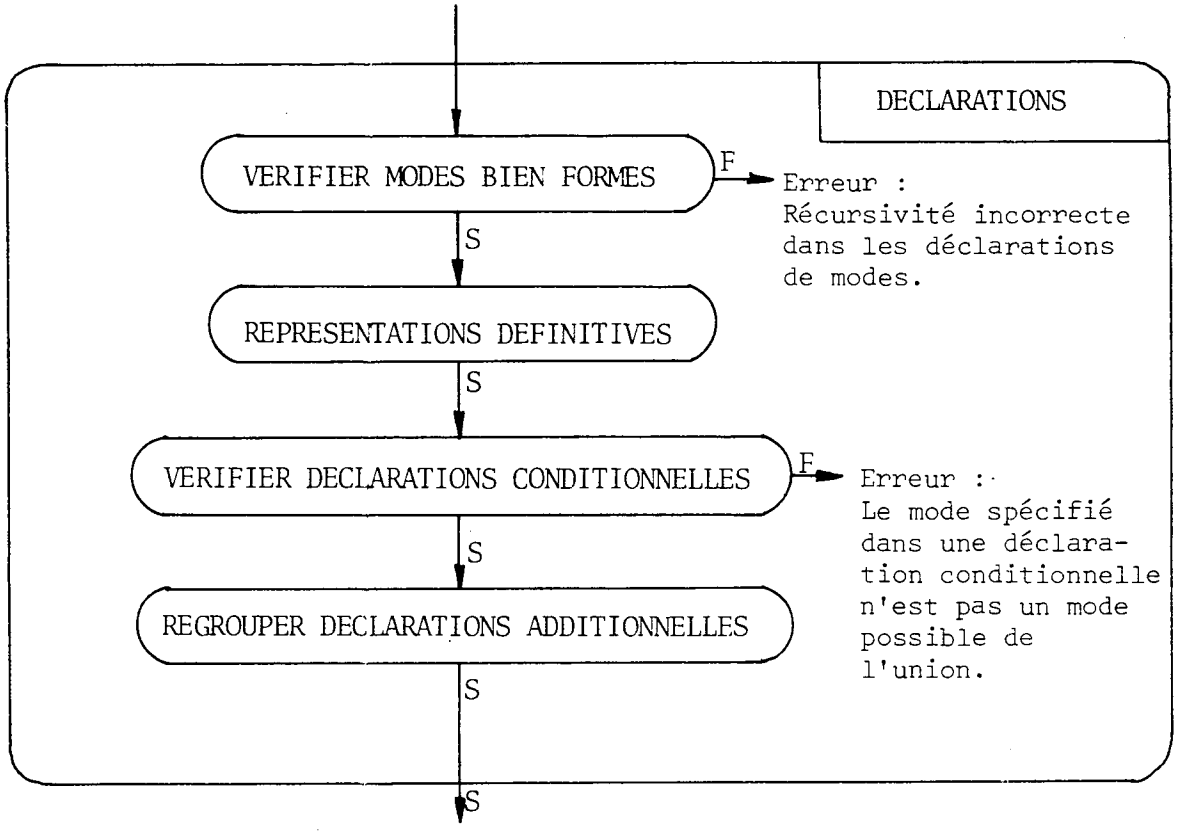




Note : La procédure "MEME MODE" n'est pas décrite ici : elle est en fait une transposition à peu près directe en AMBIT/G de l'algorithme qui a été donné au paragraphe E.2.3. pour définir la relation d'égalité entre deux modes.


VIDER BRANCHES
P
Q
R
T
P
S
S
P
F
S
P
S
P
F
P
Q
S
S
notation
R
S
DERNIER (R)
S
DERNIER (Q)
S
Q
R


DERNIER (P)
P
S
S
F
S
P

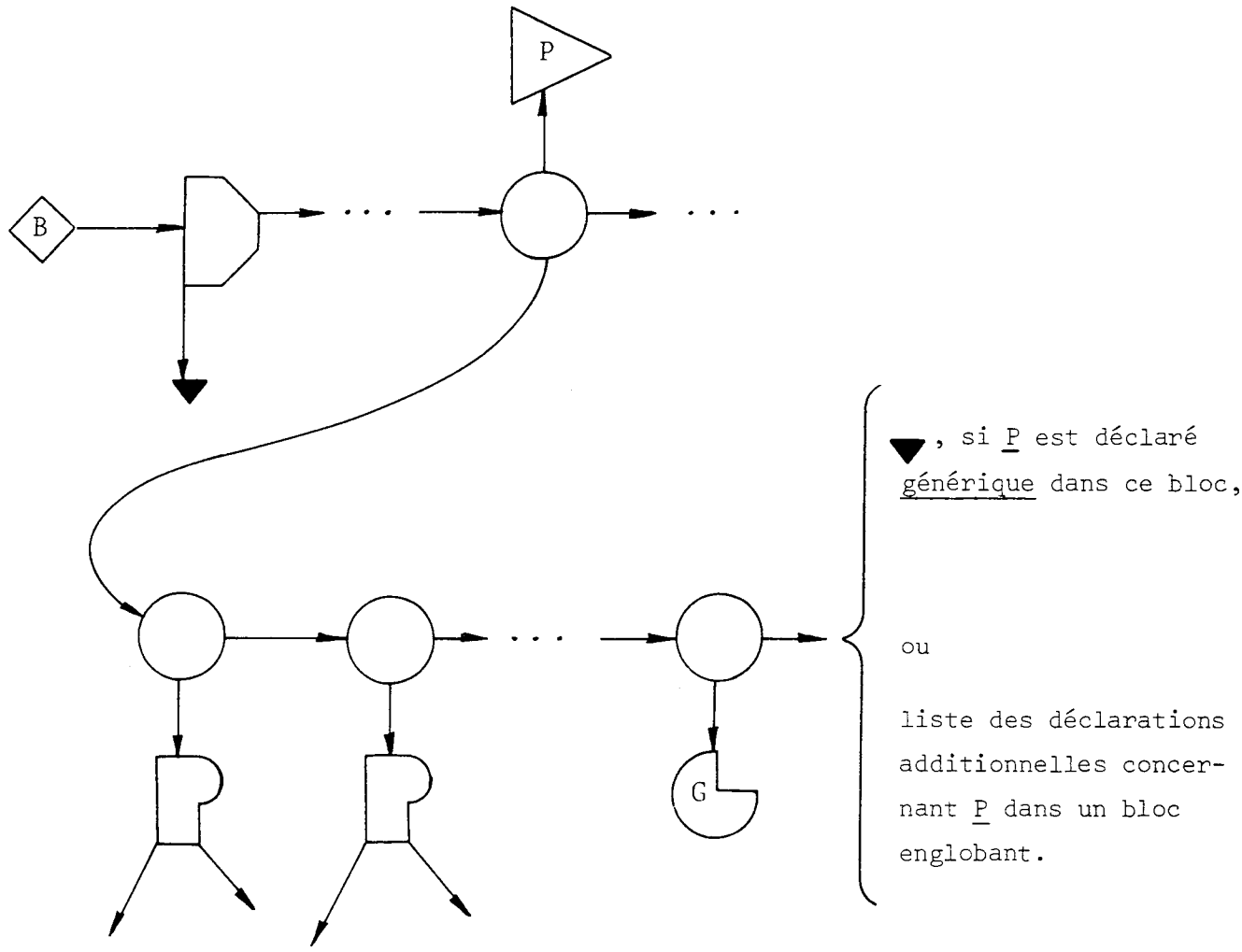


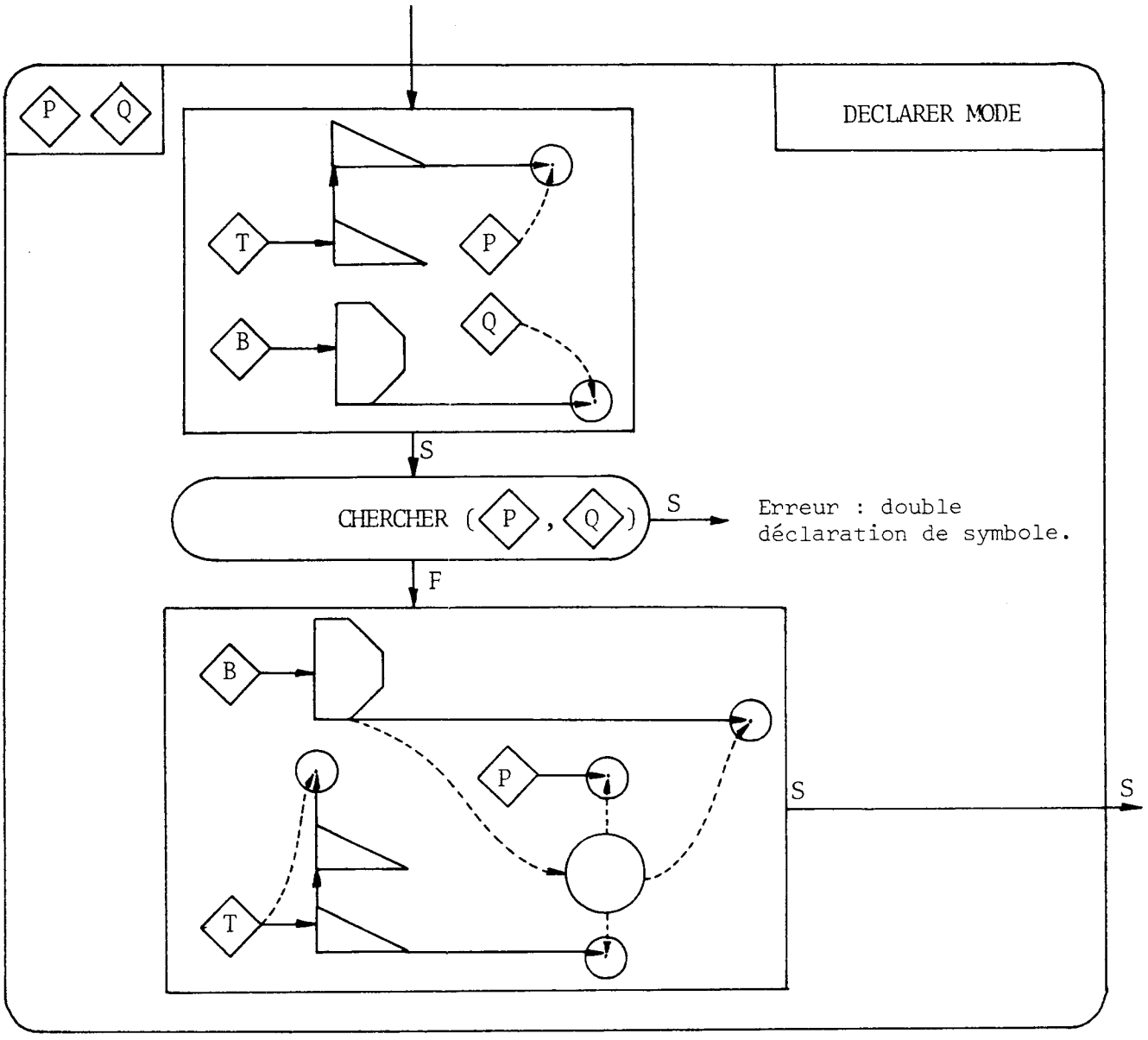
- Procédure "VERIFIER MODES BIEN FORMES" : lors de l'appel de cette procédure, les modes déclarés sont sous la forme de leurs représentations "temporaires". Cette procédure vérifie que tous les modes déclarés sont "bien formés", selon le critère défini au paragraphe E.2.2.

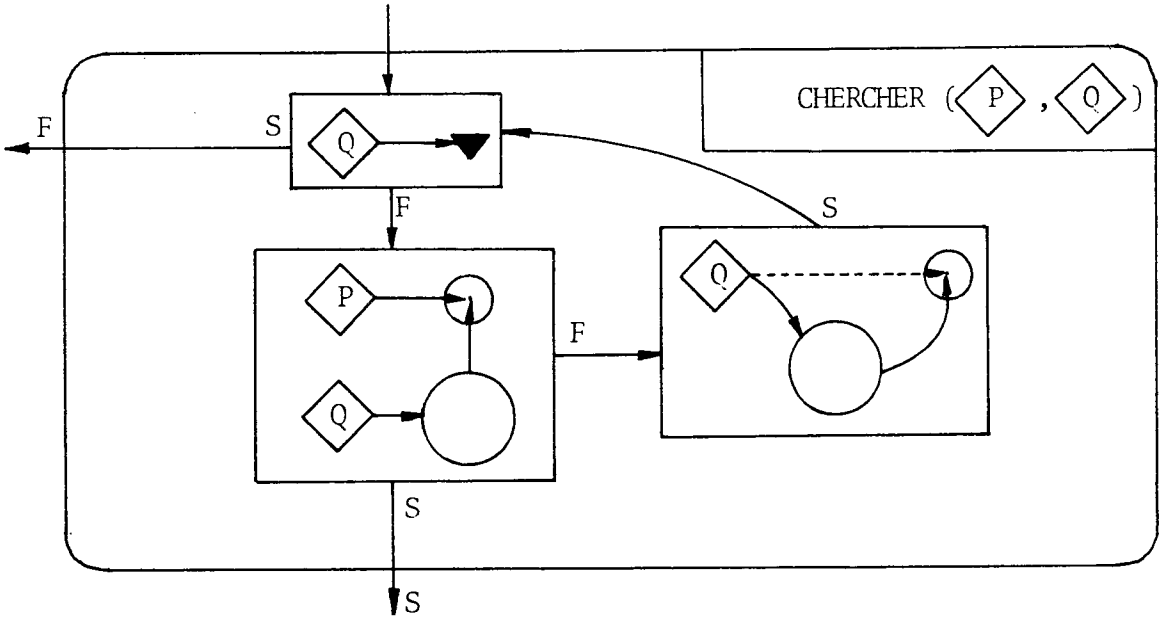
- Procédure "REPRESENTATIONS DEFINITIVES" : cette procédure remplace les représentations temporaires de tous les modes écrits dans la partie déclaration par leur représentation "définitive", telle que définie au paragraphe E.2.2.

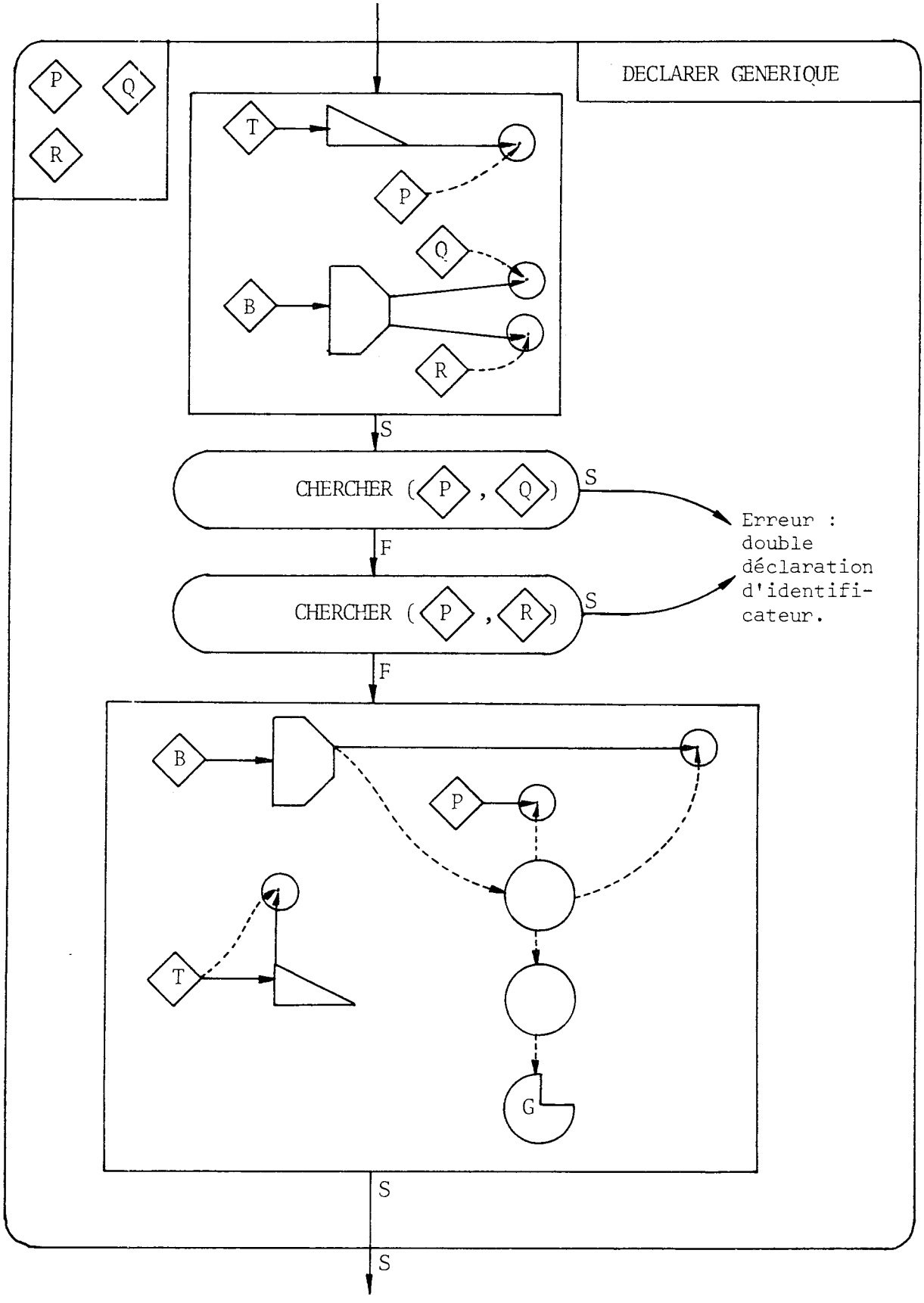
- Procédure "VERIFIER DECLARATIONS CONDITIONNELLES" : cette procédure vérifie que pour chaque déclaration conditionnelle, le mode spécifié est bien égal à l'un des modes de l'objet de mode union utilisé dans la déclaration.

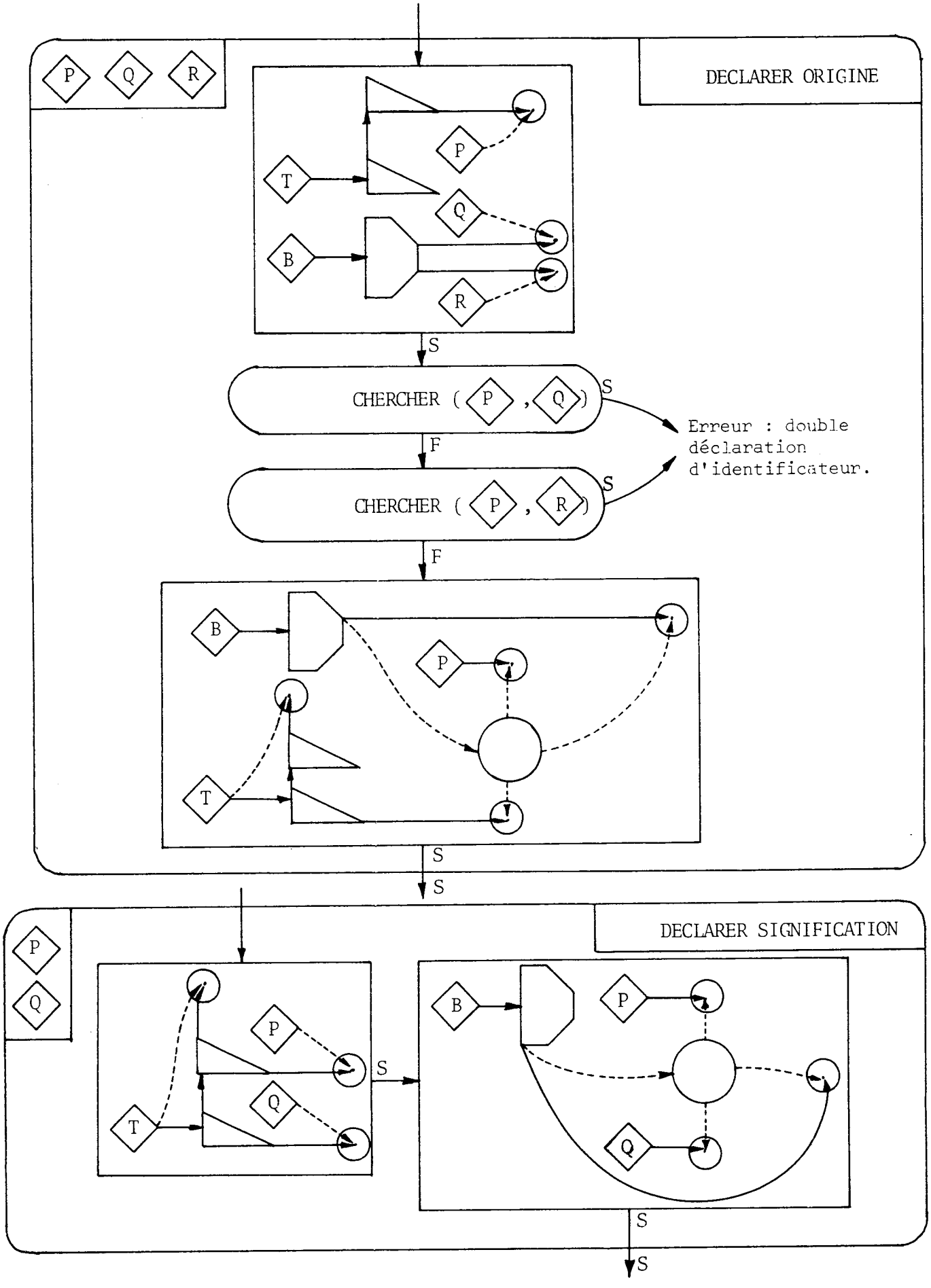
-- Procédure "REGROUPER DECLARATIONS ADDITIONNELLES" : pour chaque procédure générique P cette procédure regroupe en une liste les déclarations additionnelles qui la concernent, et insère cette liste dans la liste des déclarations d'origines :

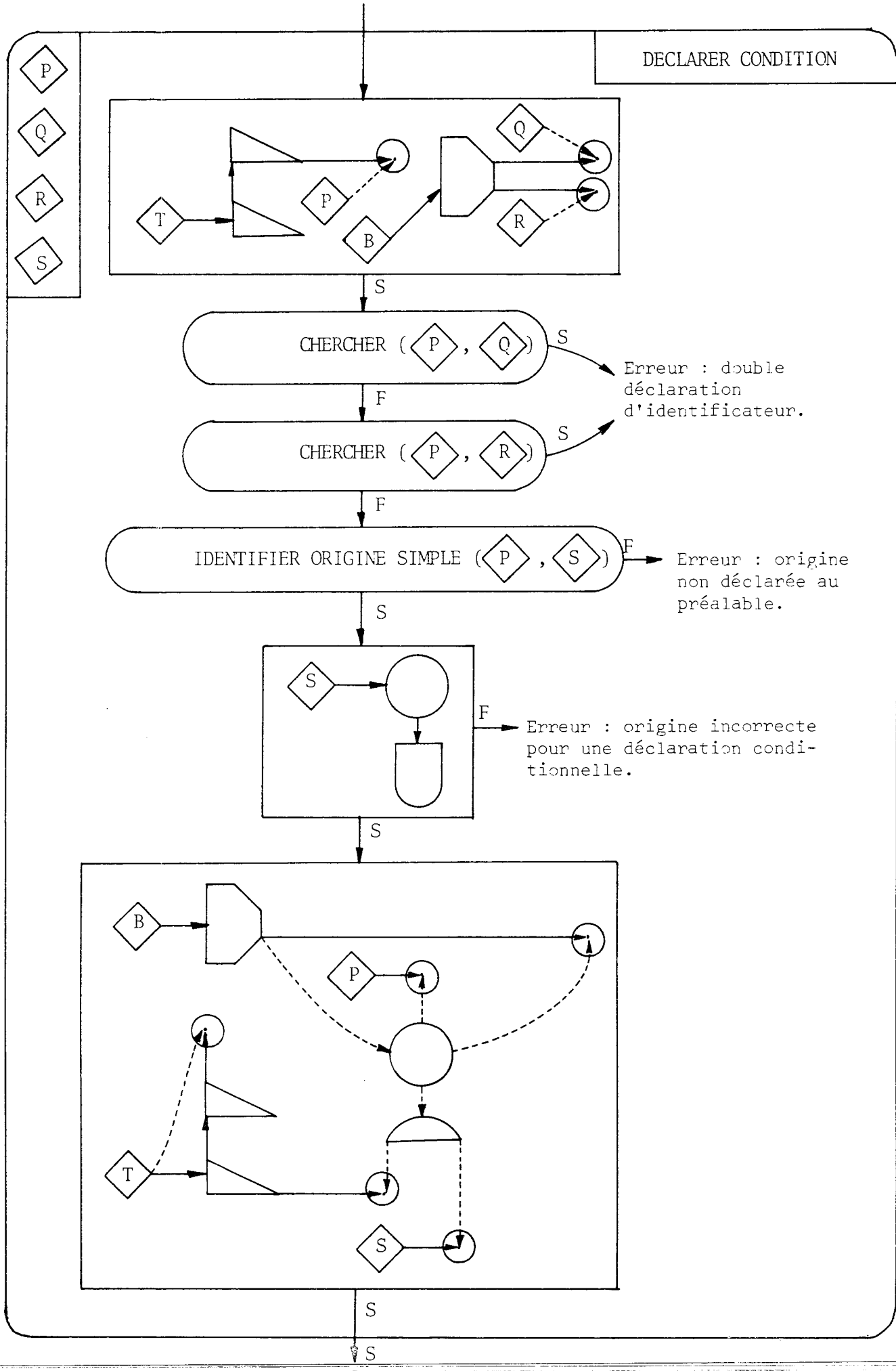


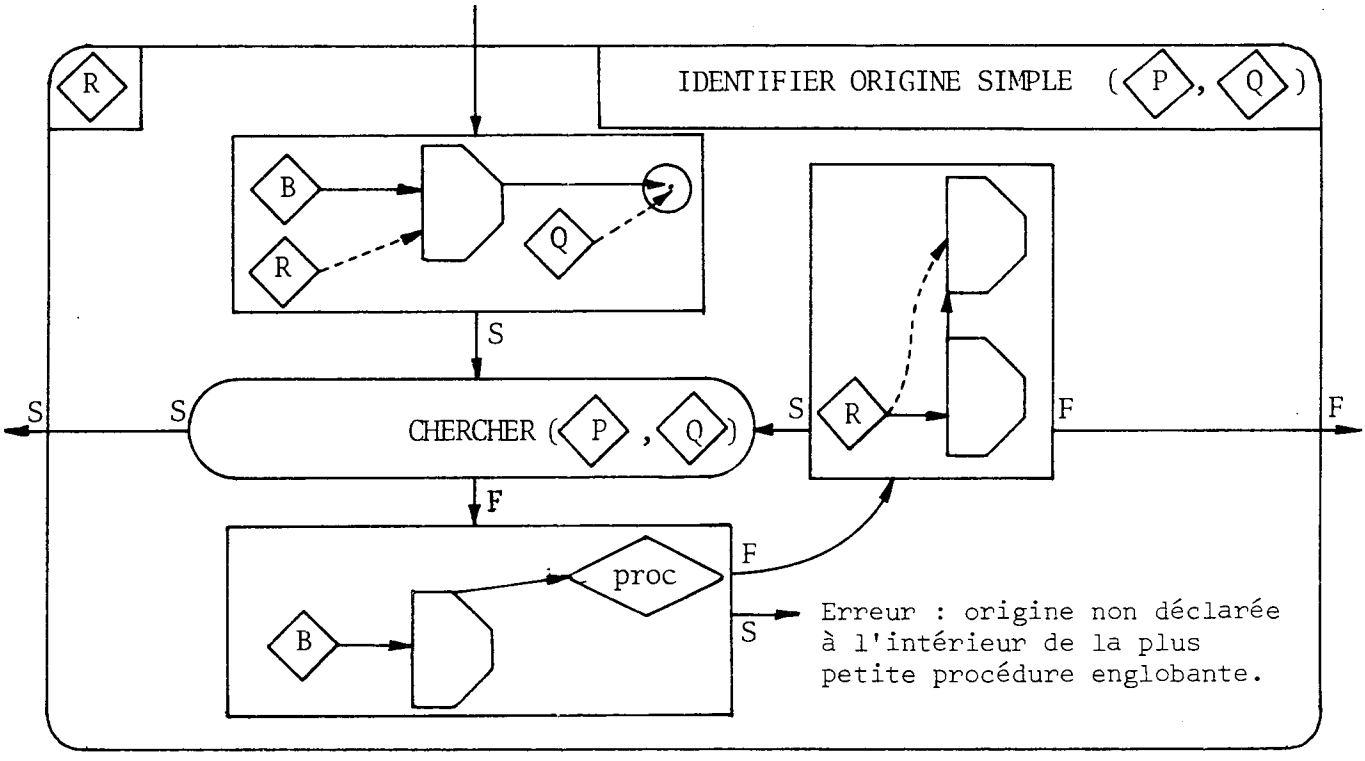

P
Q
DECLARER MODE
T
P
B
Q
S
CHERCHER (P , Q)
S
Erreur : double
déclaration de symbole.
F
B
P
S
S
T

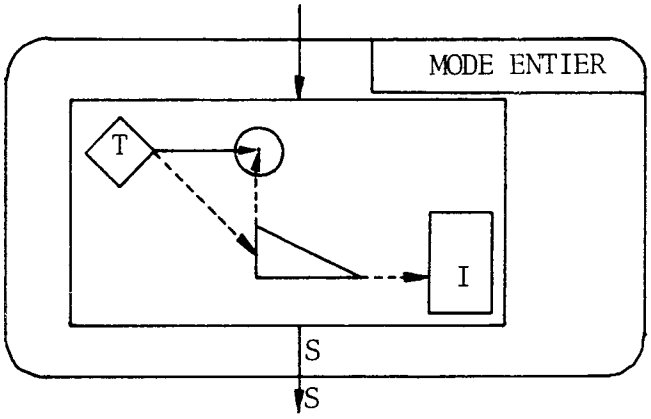

CHERCHER (P , Q)
F
S
Q
F
S
P
Q
F
Q
S
S

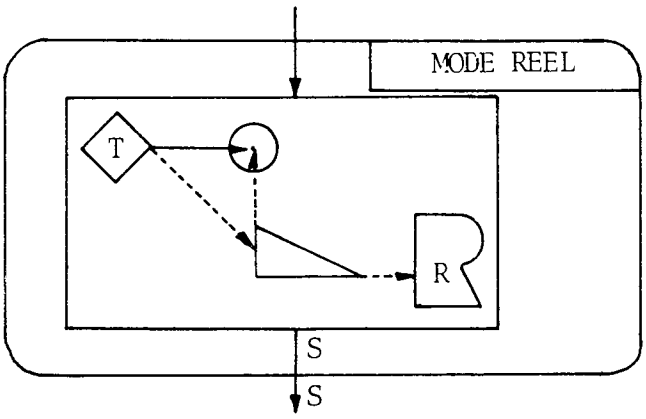

P
Q
R
DECLARER GENERIQUE
T
P
Q
B
R
S
CHERCHER (P , Q)
S
Erreur : double déclaration d'identificateur.
F
CHERCHER (P , R)
S
F
B
P
T
G
S
S

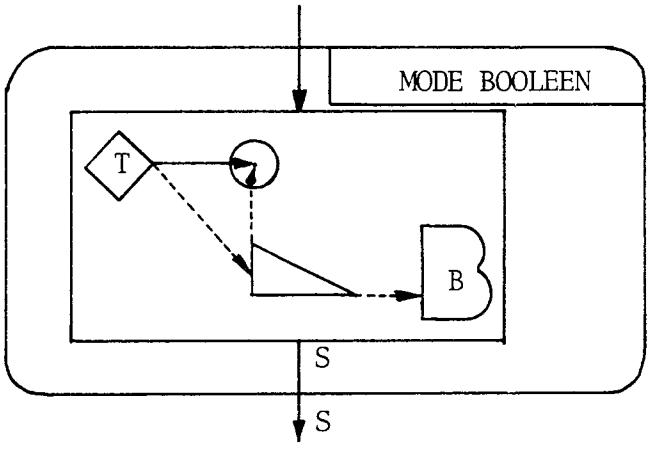

P
Q
R
DECLARER ORIGINE
T
B
S
CHERCHER (P , Q)
Erreur : double
déclaration
d'identificateur.
F
CHERCHER (P , R)
DECLARER SIGNIFICATION

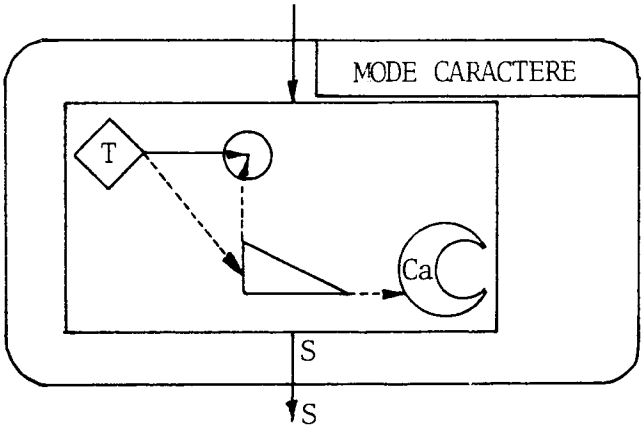

DECLARER CONDITION
P
Q
R
S
T
B
S
CHERCHER (P , Q)
S
Erreur : double déclaration d'identificateur.
F
CHERCHER (P , R)
S
F
IDENTIFIER ORIGINE SIMPLE (P , S)
F
Erreur : origine non déclarée au préalable.
S
S
F
Erreur : origine incorrecte pour une déclaration conditionnelle.
S
B
P
T
S
S
S

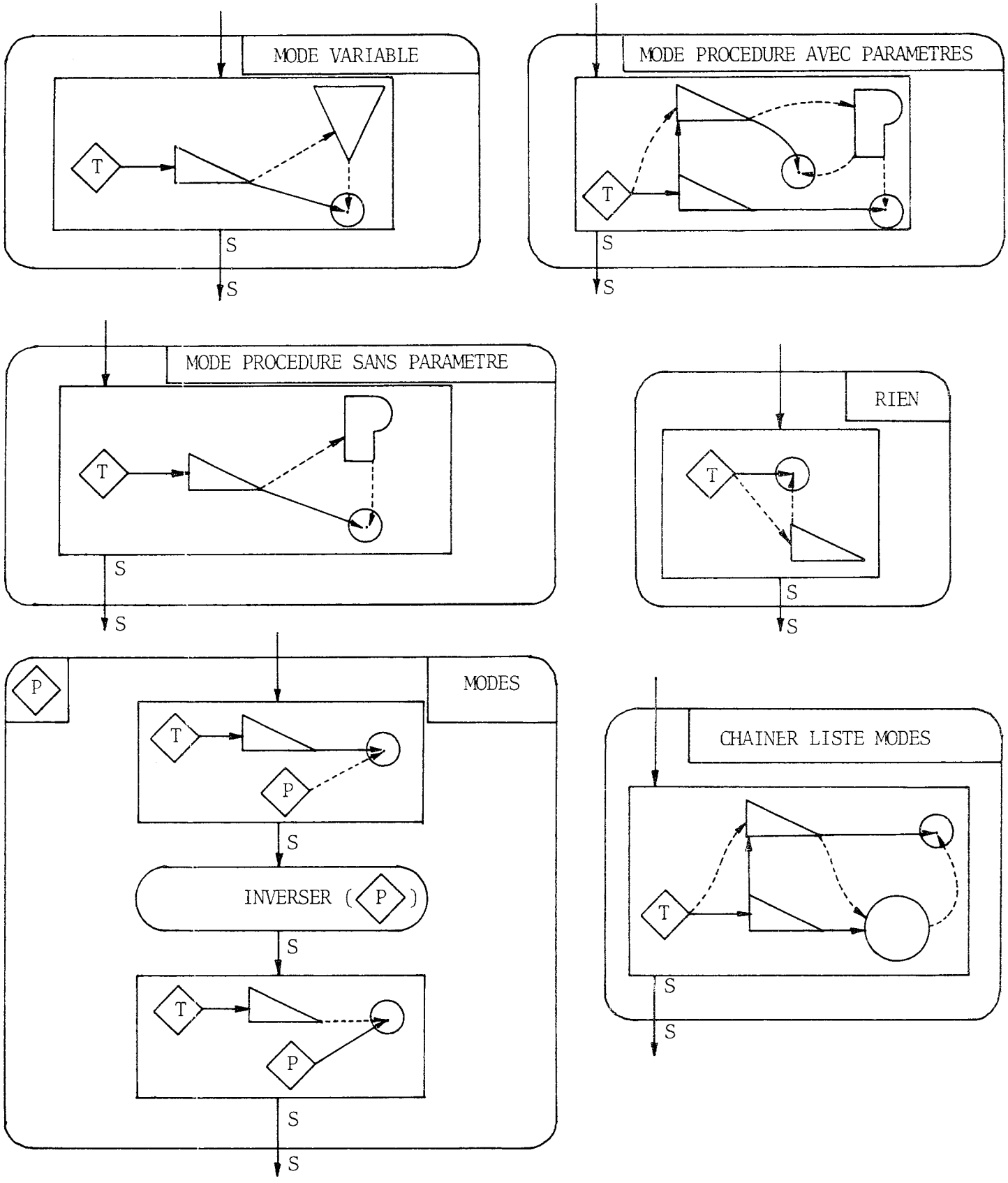

R
IDENTIFIER ORIGINE SIMPLE (P , Q)
B
R
Q
S
S
S
CHERCHER (P , Q)
S
R
F
F
F
proc
B
F
S
Erreur : origine non déclarée
à l'intérieur de la plus
petite procédure englobante.


MODE ENTIER
T
I
S
S

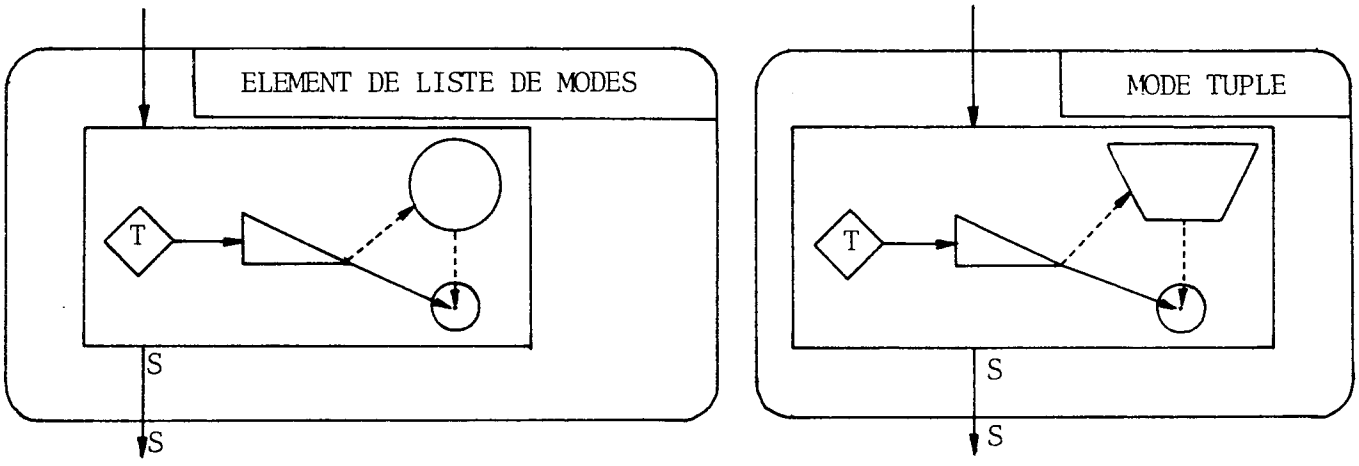

MODE REEL
T
R
S
S


MODE BOOLEEN
T
B
S
S

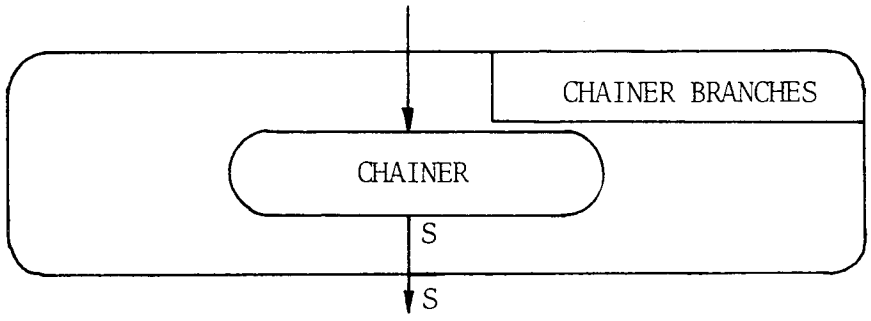

MODE CARACTERE
T
Ca
S
S

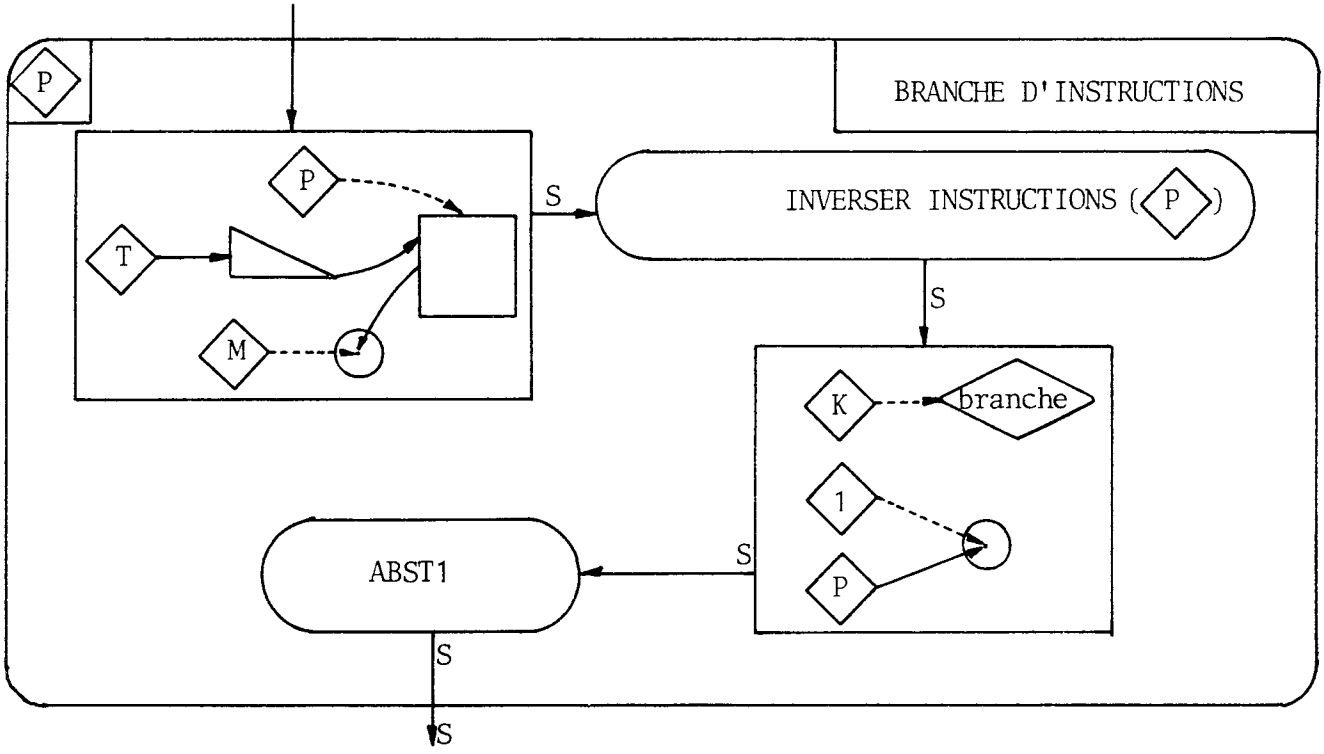


Note : la procédure "INVERSER" a été définie plus haut, au paragraphe 3.1.3.7.



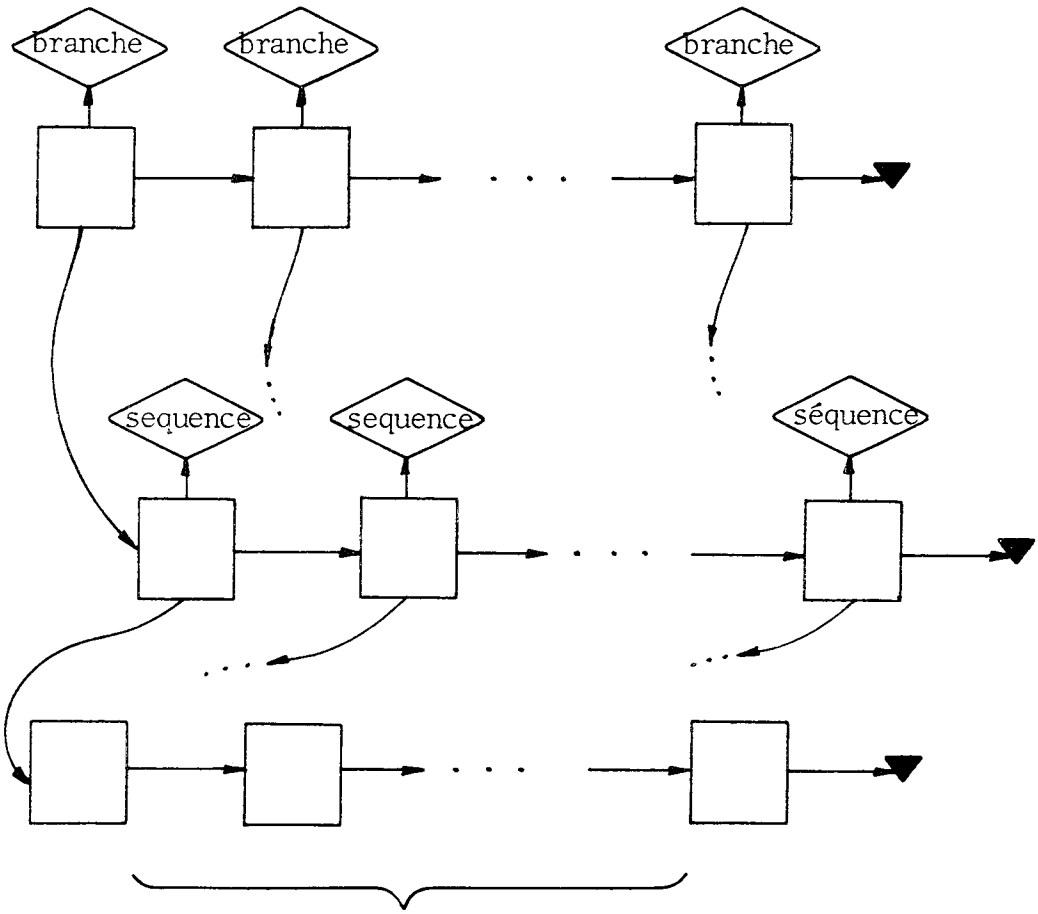
Etc... Les autres procédures qui construisent les représentations temporaires des modes fonctionnent selon des méthodes analogues à celles qui viennent d'être décrites.





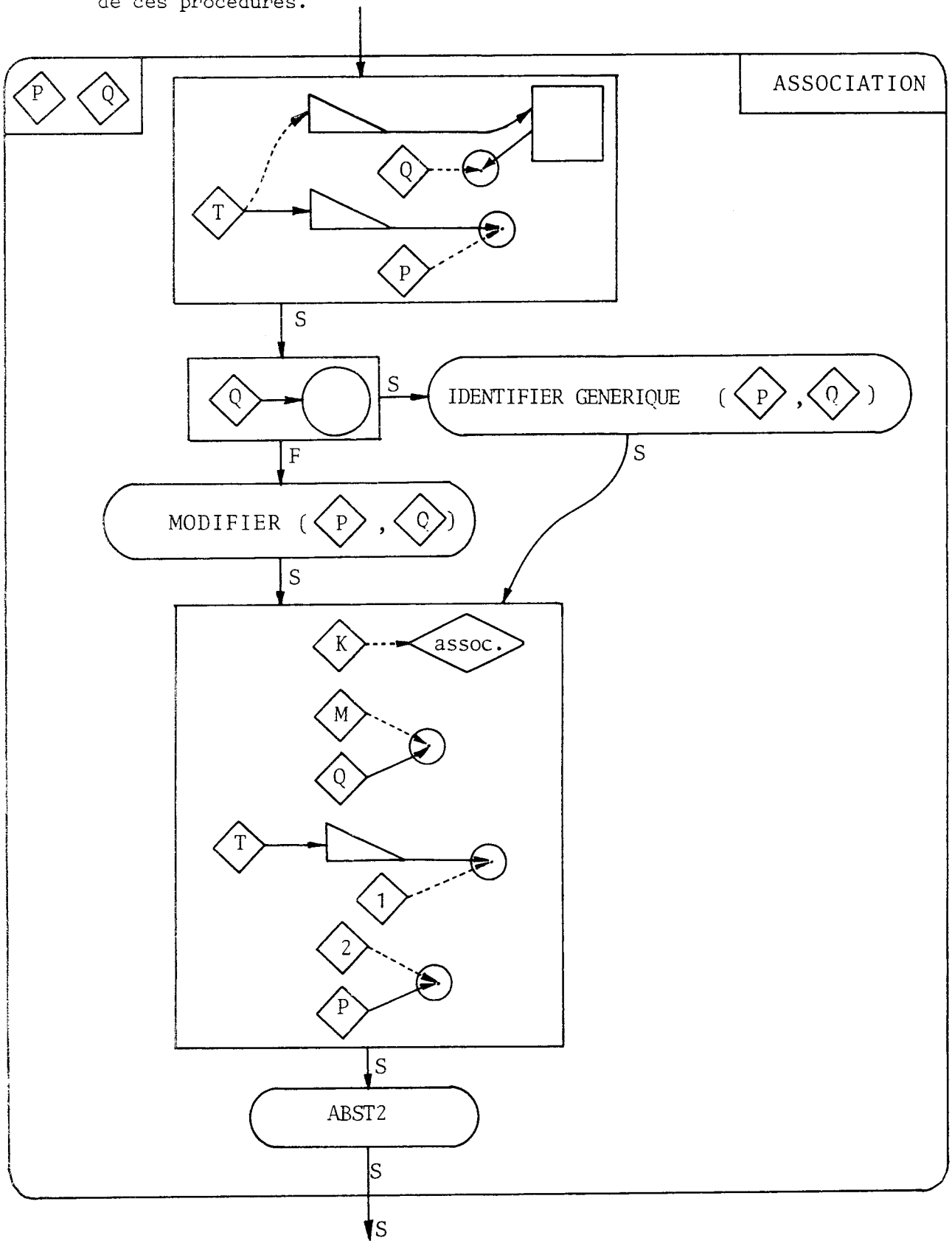
Les procédures suivantes : CHAINER SEQUENCES,
SEQUENCE D'INSTRUCTIONS,
CHAINER INSTRUCTIONS,

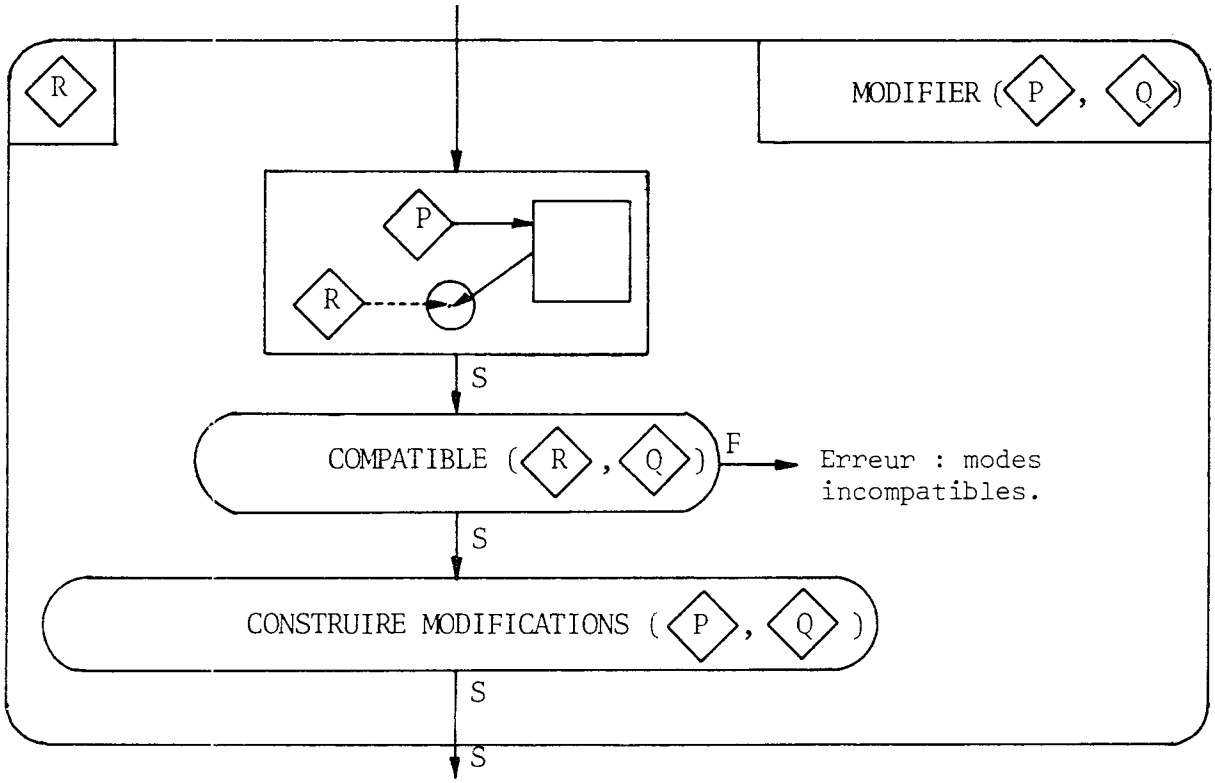
construisent la structure de la partie instruction d'un bloc :



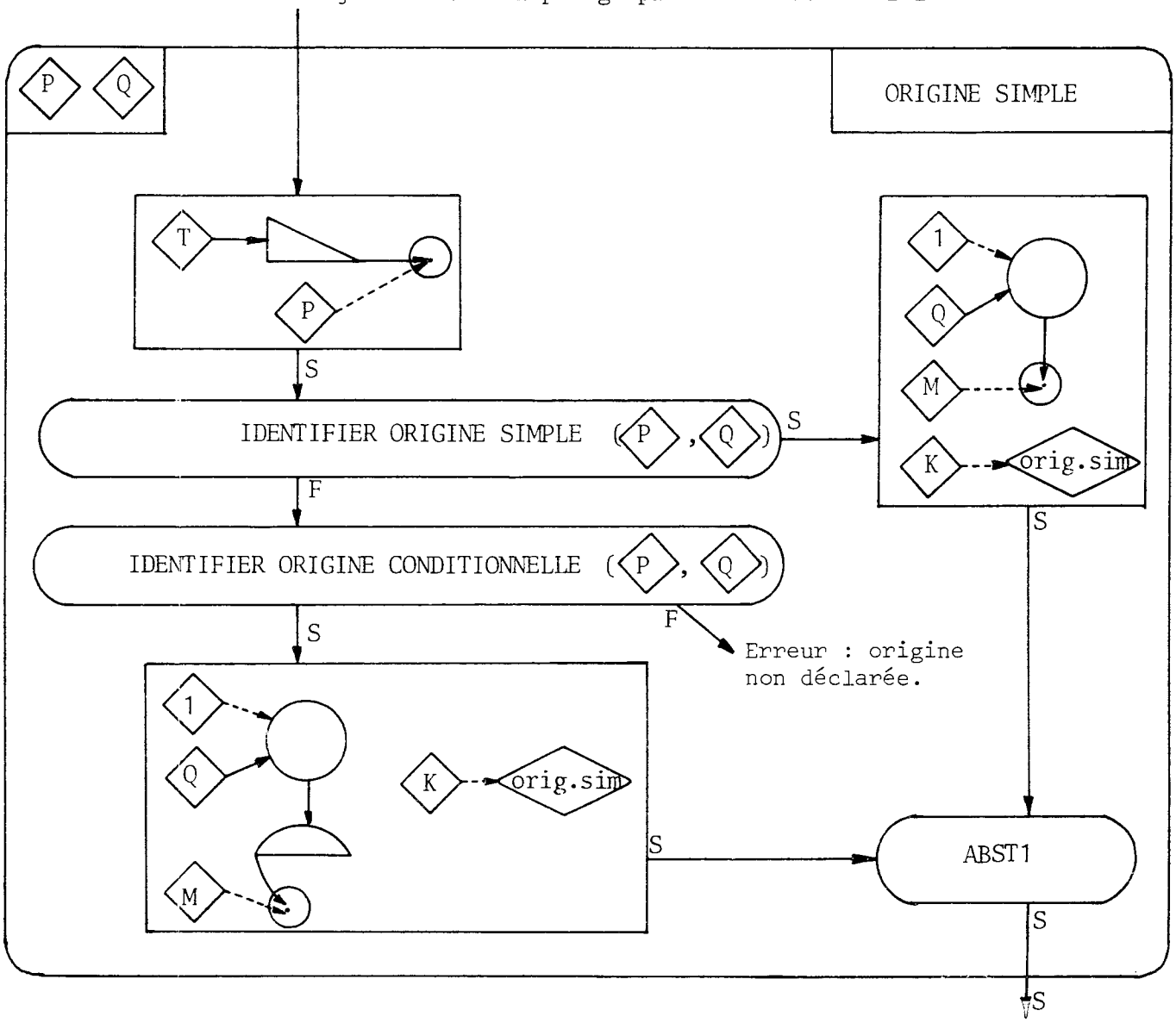
- la dernière instruction a éventuellement subi des modifications pour avoir un résultat de mode bool ;
- les autres instructions ont éventuellement été modifiées par des suppressions de valeur entraînant des extractions de valeur et des appels implicites.

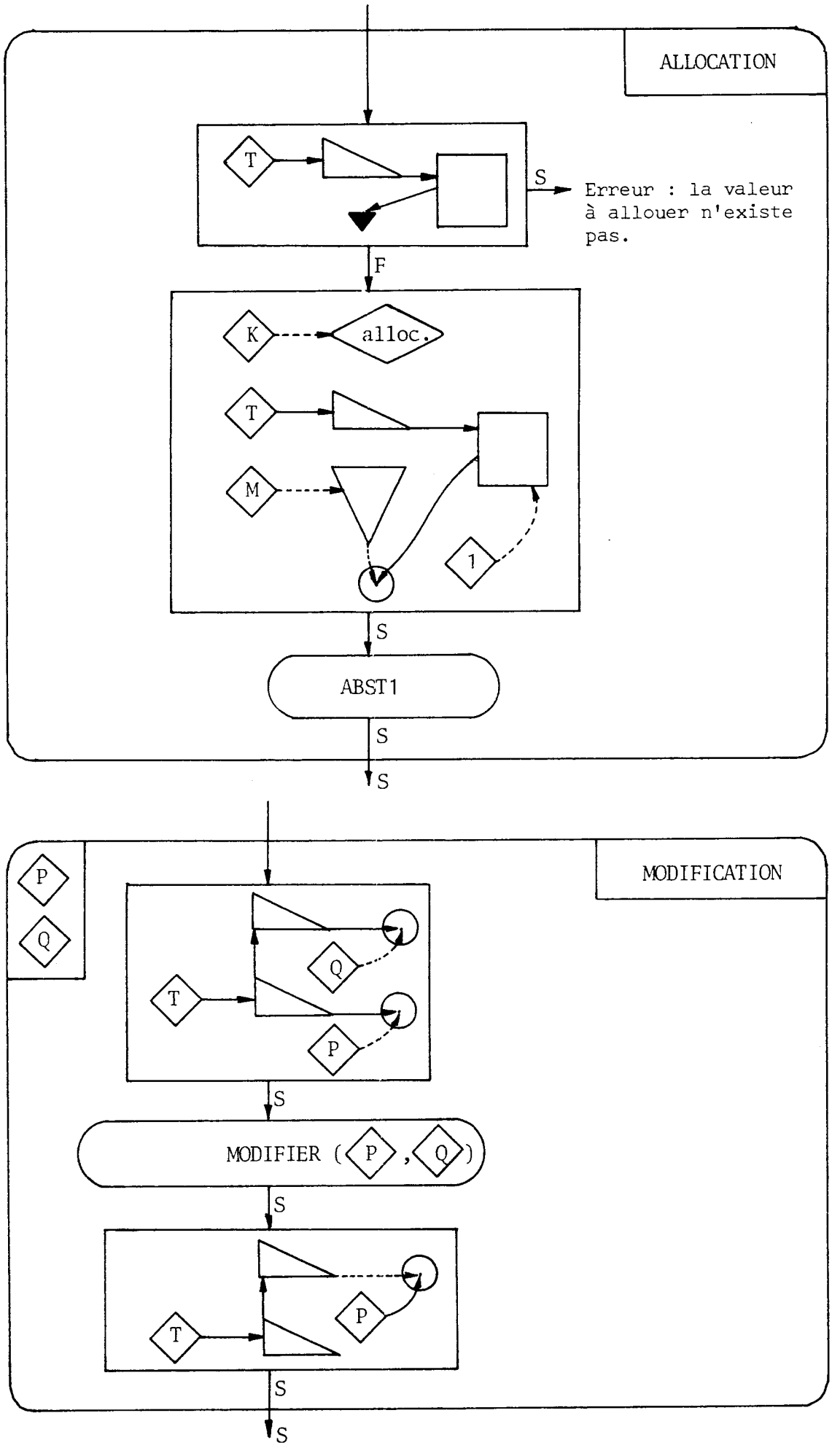
Dans ce qui suit, toutes les procédures fonctionnent selon le même principe général : construction d'une nouvelle instruction dont les constituants sont au sommet de la pile ◇T◇ . On ne donne ici que quelques unes de ces procédures.

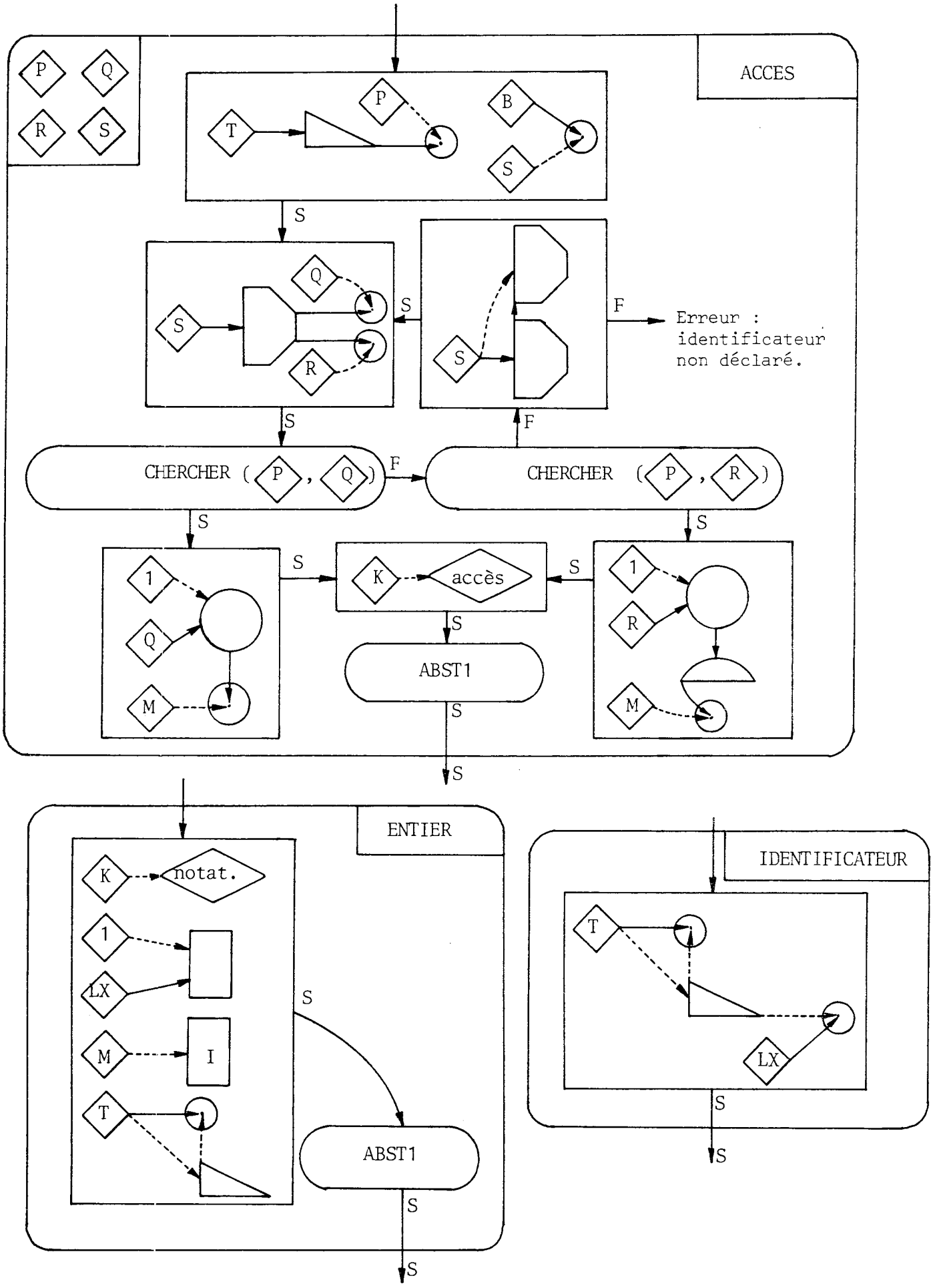




Note : les procédures "COMPATIBLE" et "CONSTRUIRE MODIFICATIONS" expriment en AMBIT/G les mécanismes déjà décrits aux paragraphes E.2.4 et F.2.1.1.



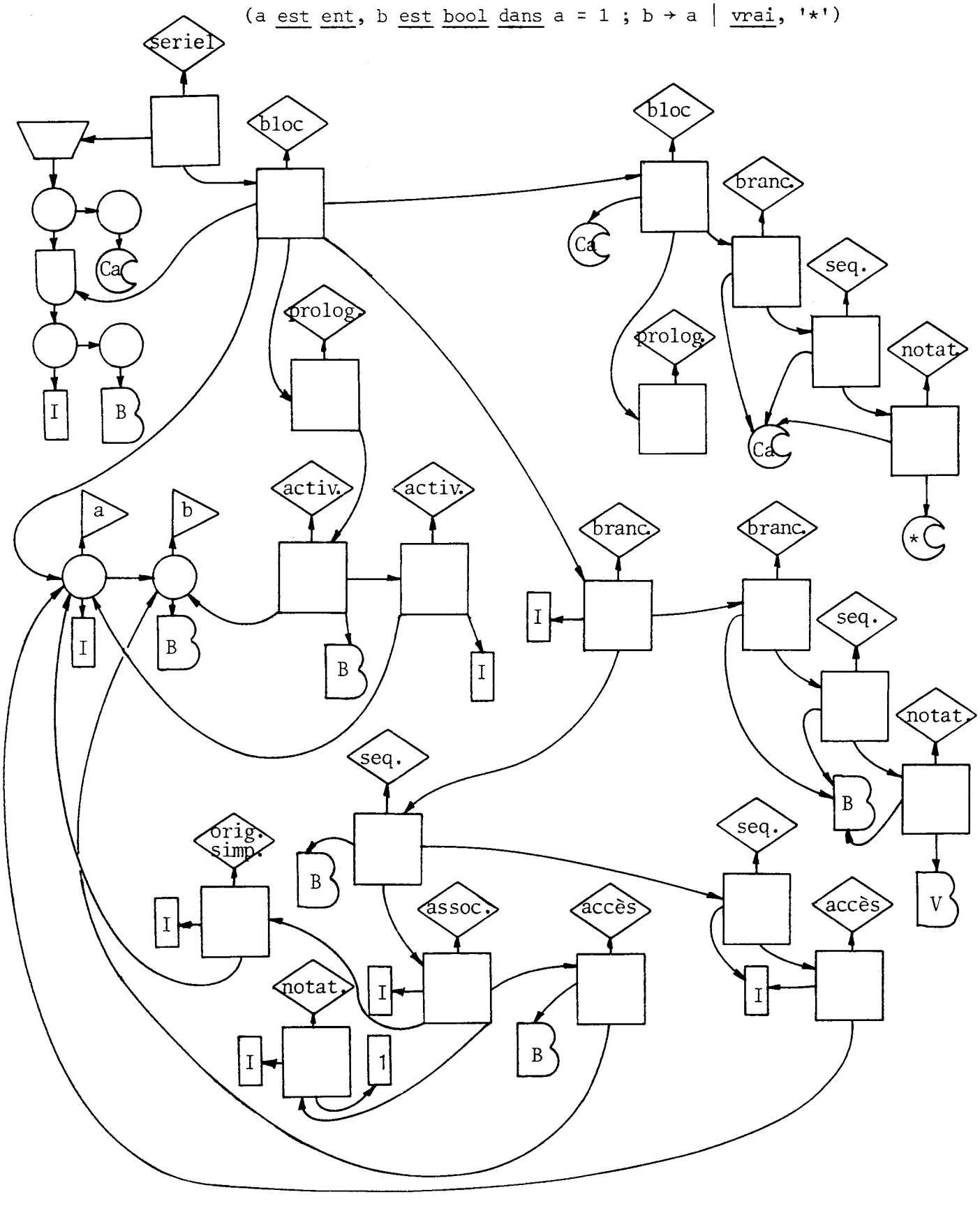

ALLOCATION
T
S
Erreur : la valeur à allouer n'existe pas.
F
K
alloc.
T
M
1
S
ABST1
S
S
P
Q
MODIFICATION
Q
T
P
S
MODIFIER (P , Q)
S
P
T
S
S


ACCES
P
Q
R
S
T
B
S
F
Erreur : identificateur non déclaré.
CHERCHER (P , Q)
CHERCHER (P , R)
1
K
accès
ABST1
M
ENTIER
notat.
LX
I
IDENTIFICATEUR

Exemple de représentation d'une expression composée sérielle construite par le traducteur :

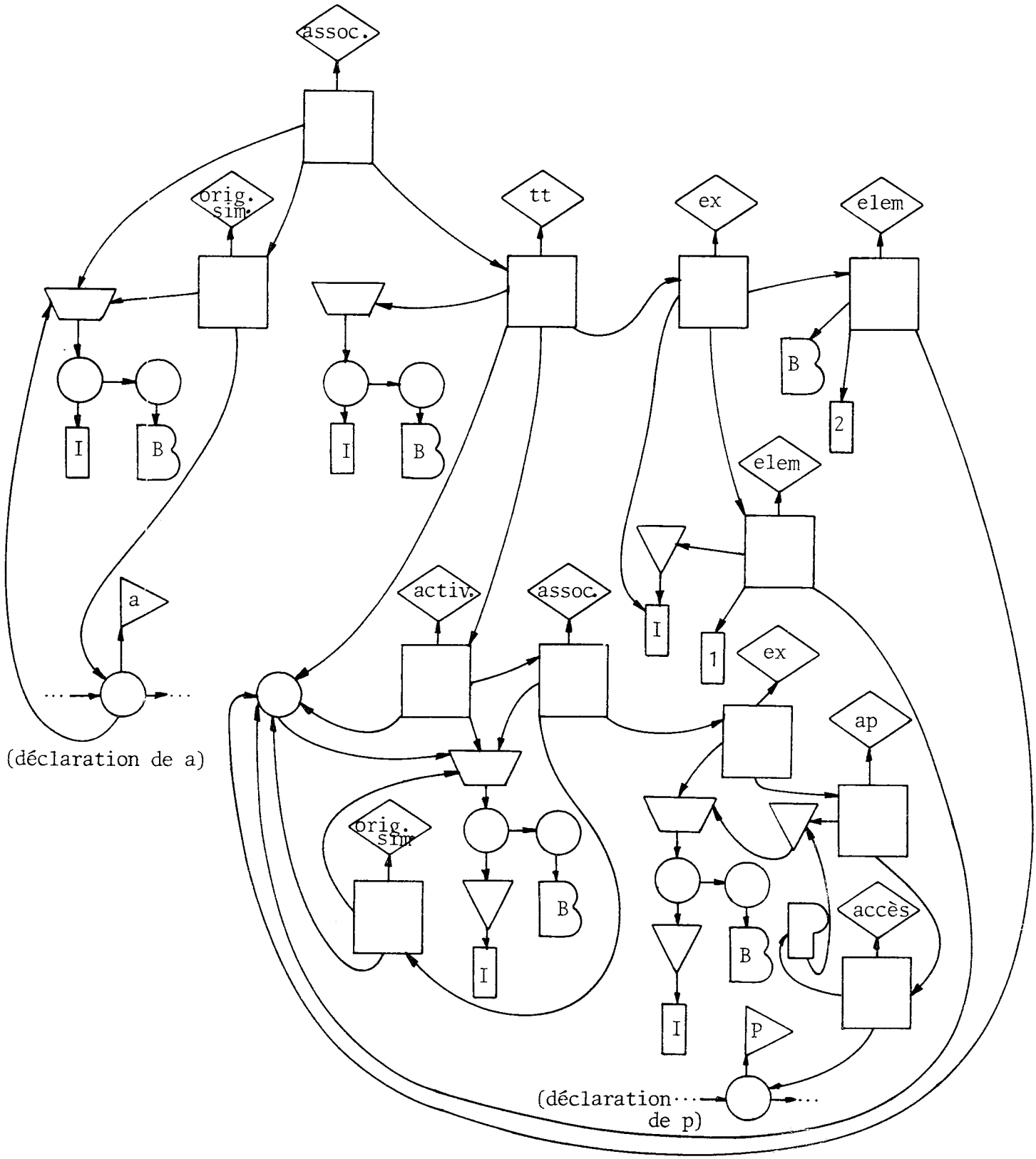
(les arcs aboutissant au vide ne sont pas représentés et certains objets sont dessinés plusieurs fois, pour plus de clarté).

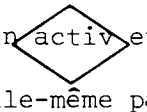
(a est ent, b est bool dans a = 1 ; b → a | vrai, '*')

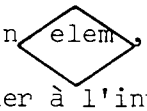


Exemple de représentation abstraite d'une instruction comportant des modifications.

Dans la portée de : a est tuple (ent, bool)
p est proc var tuple (var ent, bool)
l'association : a = p
sera représentée par :



Sur cet exemple, on remarque la façon générale dont opèrent les modifications composées (tt par exemple) : elles utilisent une origine auxiliaire à laquelle est associée la valeur (structure ou tuple) de départ. (Elles créent cette origine par une opération 

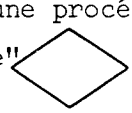
et lui associent la valeur de départ, calculée éventuellement elle-même par des modifications). Elles utilisent également l'opération 

, elle aussi connue seulement du programme abstrait, pour accéder à l'intérieur d'un tuple ou d'une structure. Les composants de la valeur finale sont obtenus, les uns après les autres, par modification des éléments successifs de la valeur associée à l'origine auxiliaire. La valeur finale est recomposée ensuite par l'opération elle-même (tt par exemple, si c'est un tuple que l'on doit obtenir).

3.1.4.2. Description d'une partie de l'interpréteur

a. Fonctionnement général de l'interpréteur

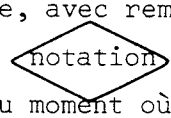
L'ensemble de l'interpréteur est constitué de procédures qui fonctionnent toutes selon un principe unique :

- Il y a une procédure par "type" d'instruction, indiqué par le "losange"  attaché à chaque construction du programme abstrait.

- Chaque procédure demande d'abord l'exécution des composants de l'instruction qu'elle est chargée d'interpréter. Lorsque ces exécutions sont terminées, elle utilise les résultats obtenus pour construire le résultat de l'instruction selon des règles propres à cette instruction.

Il y a aussi quelques remarques à faire sur la définition de certains aspects particuliers de la sémantique dynamique de BASEL :

- Le résultat de toute instruction, s'il existe, est attaché à cette instruction elle-même, quand son interprétation est terminée, par un arc issu du même point-origine que celui qui indiquait le mode de cette instruction pendant la traduction.

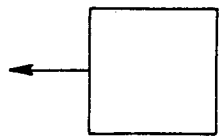
- L'évaluation des textes de procédures est faite par recopie du texte lui-même, avec remplacement des accès globaux par des expressions de type  , où la valeur "notée" est celle associée à l'origine au moment où s'effectue l'évaluation du texte.

- Les appels de procédure sont implémentées par la construction effective de l'expression composée à un seul bloc, suivie de son exécution, comme indiqué dans la description donnée au paragraphe K.2.2.2. (Comme toute procédure appelée a nécessairement eu son texte évalué au préalable, aucun conflit d'identificateur ne peut avoir lieu au cours de ce processus, bien qu'une recopie y soit effectuée).

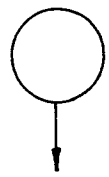
- Les procédures AMBIT/G qui effectuent les recopies de morceaux de programmes sont récursives et sont guidées, comme celles de l'interpréteur, par un parcours de la représentation abstraite. Ces procédures ne sont pas données ici, car elles n'apporteraient que très peu à la compréhension de l'ensemble.

Ainsi, à l'exception de ce qui se passe dans les actions de recopie pour définir la sémantique des procédures et de leurs appels, les seuls arcs qui soient susceptibles d'être modifiés par l'interpréteur sont :

- Ceux qui indiquent le résultat d'une instruction (modifié en fin d'interprétation de l'instruction) :

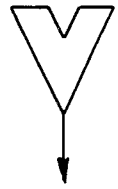


- Ceux qui donnent accès aux origines (modifiés lors de la création d'activation) :



- Ceux qui indiquent la valeur associée à une origine (modifiés par l'opération d'association) :

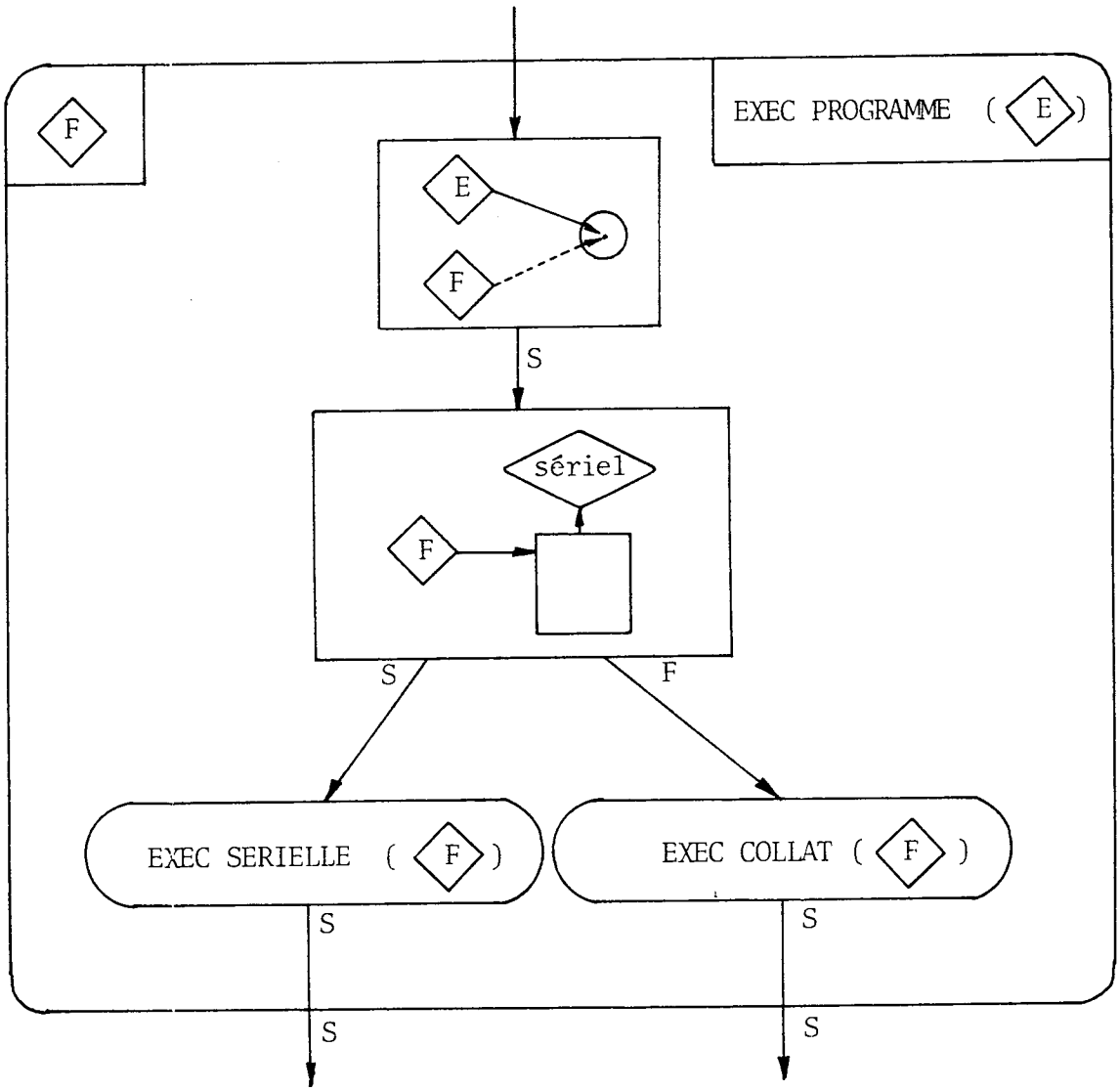
- Ceux qui indiquent la valeur retenue par une variable (modifiés par l'opération d'affectation) :

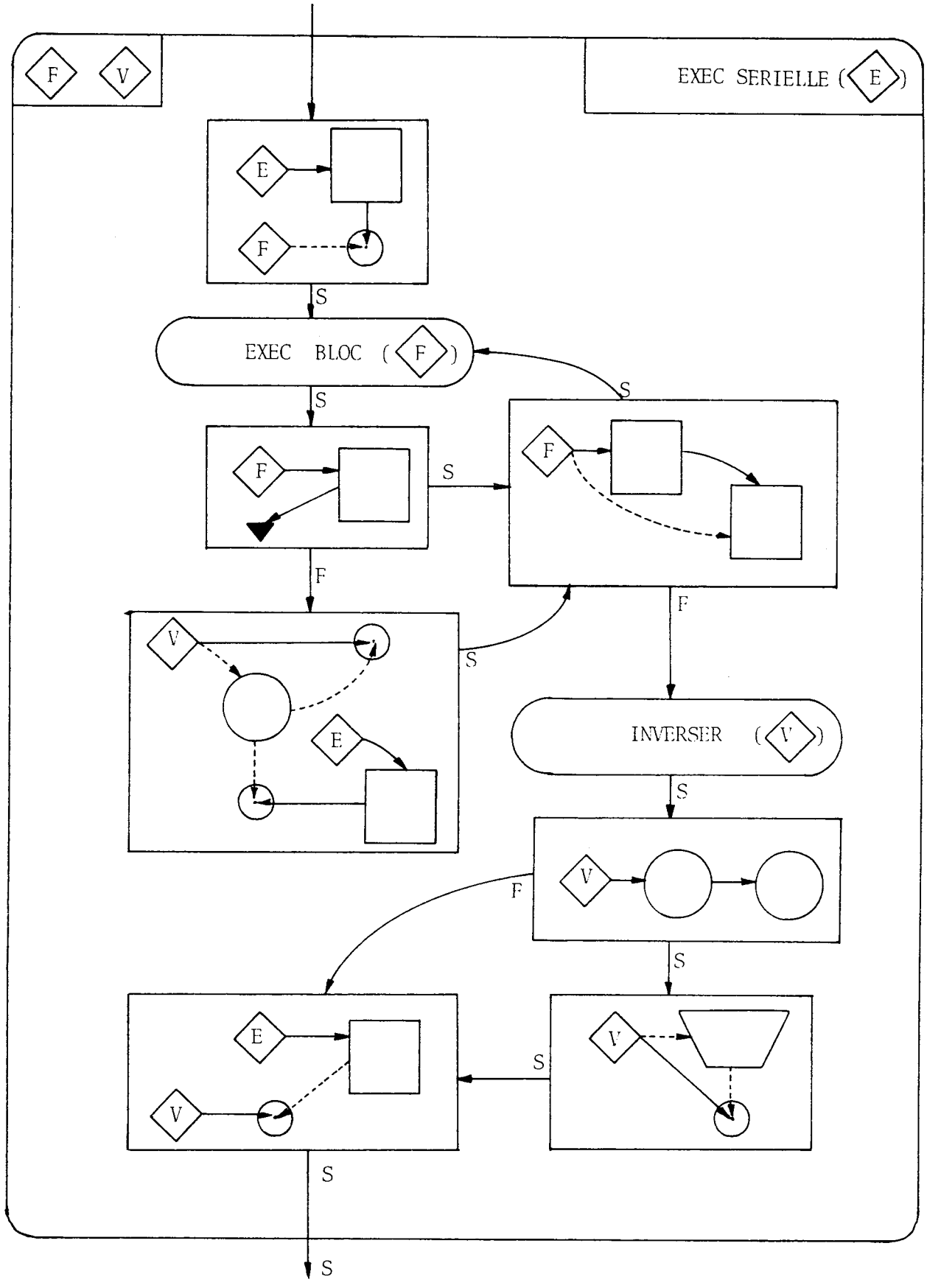


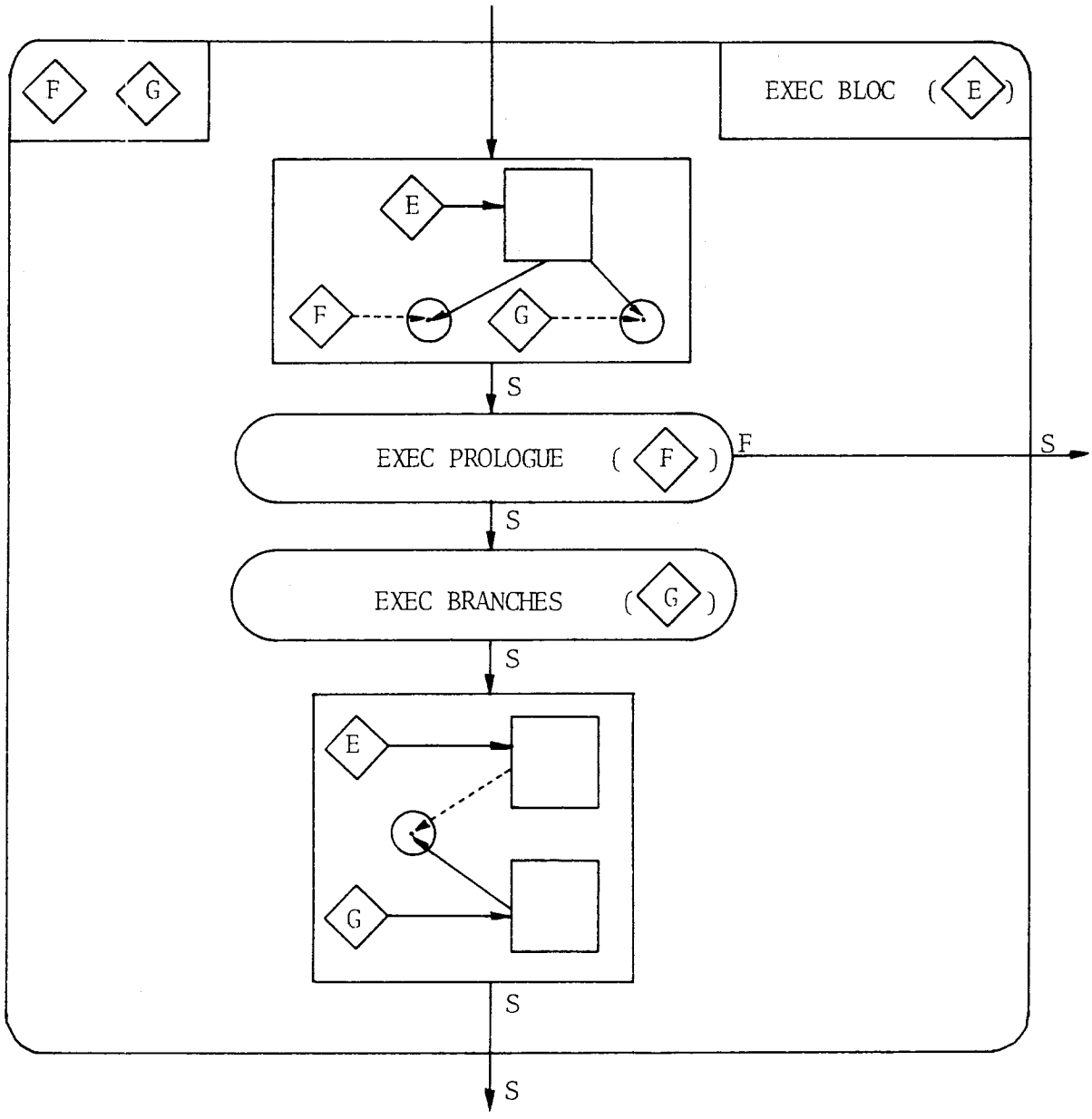
(Dans quelques cas particuliers comme l'opération de changement, d'autres arcs sont modifiés. Ceci ne sera pas détaillé ici).

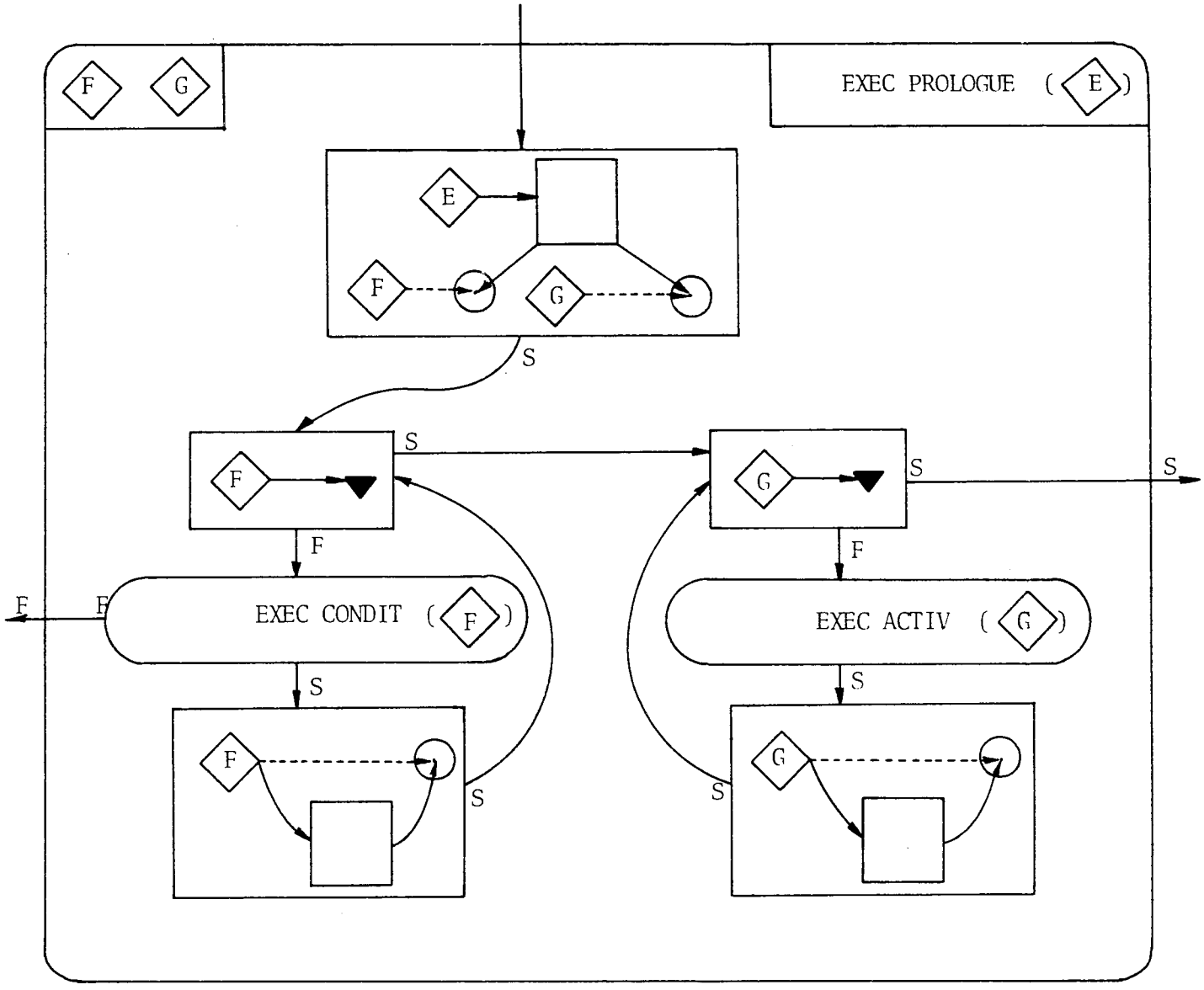
b. Quelques procédures de l'interpréteur

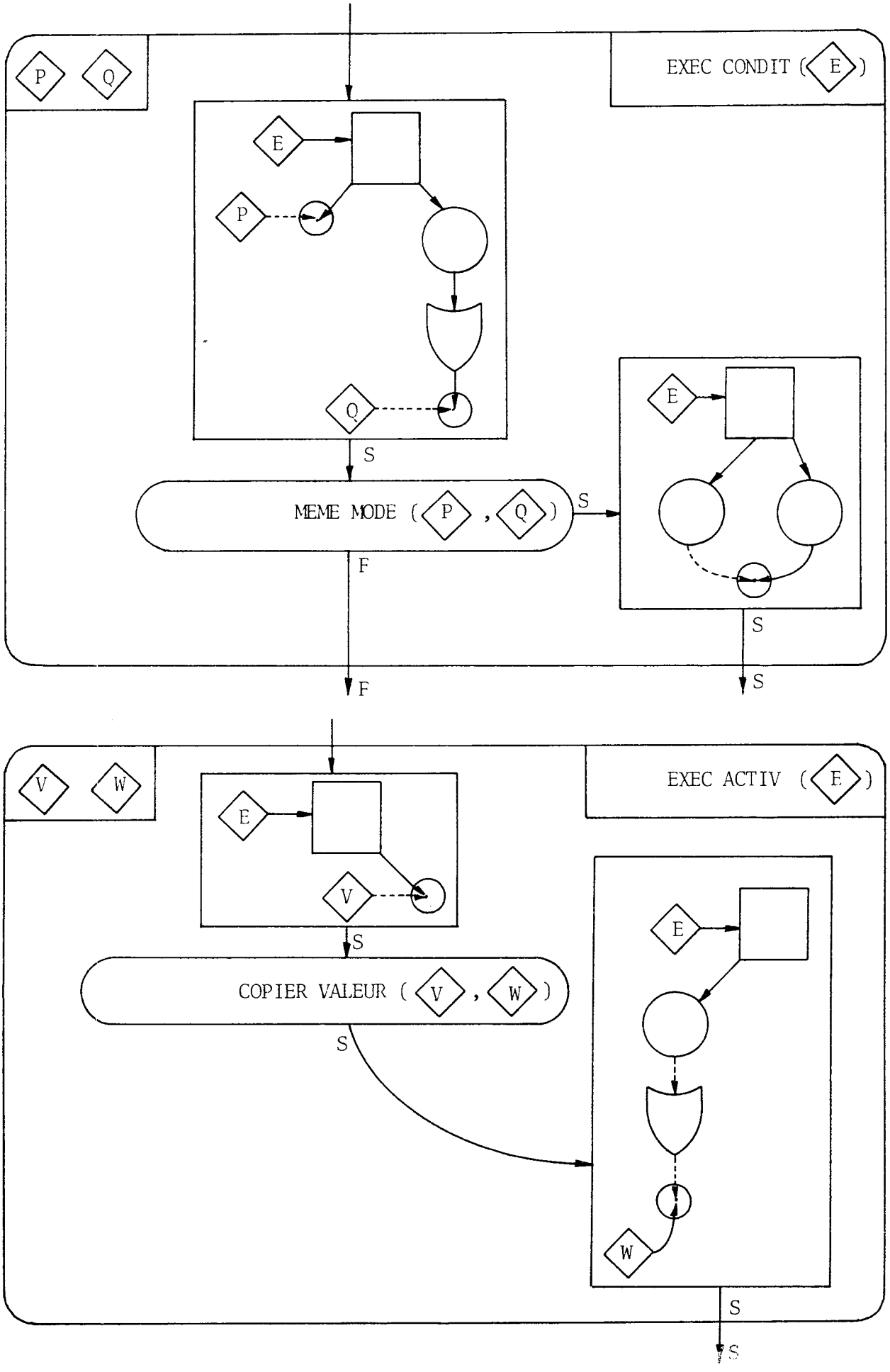
Quelques procédures de la sémantique dynamique sont données ici pour mettre en valeur le principe général du fonctionnement de l'interpréteur.

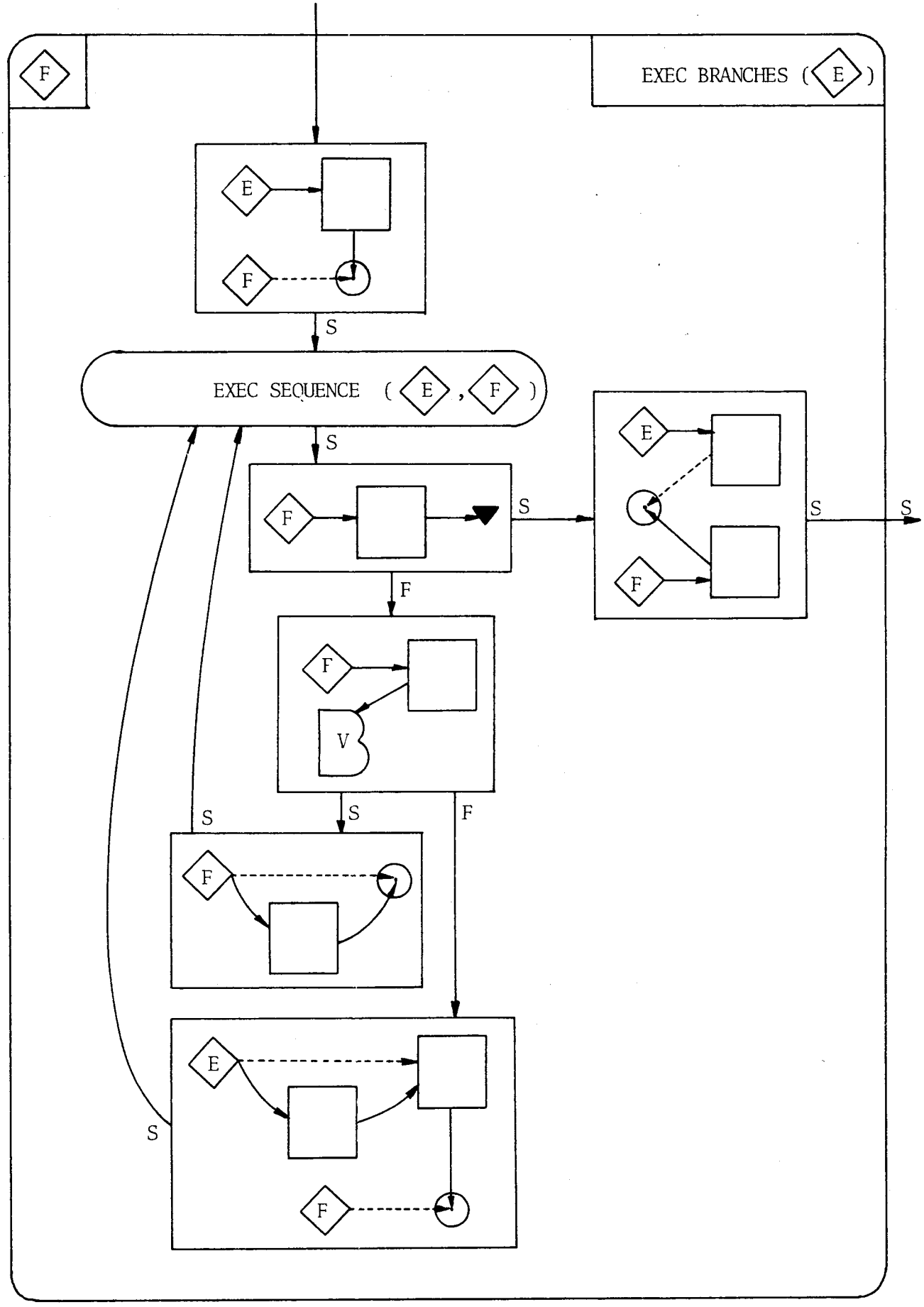


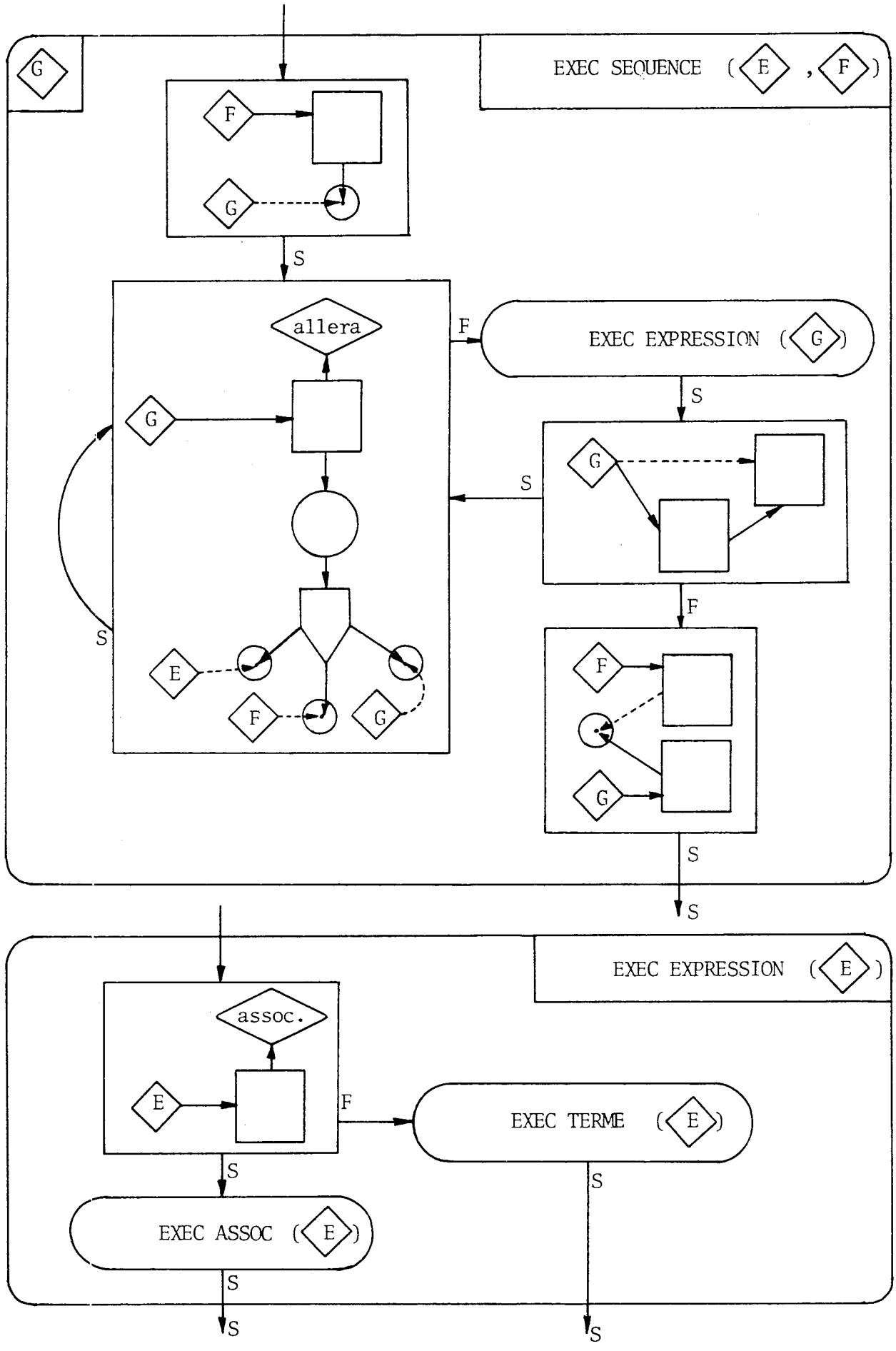

F
V
EXEC SERIELLE (E)
E
F
S
EXEC BLOC (F)
S
S
F
S
F
V
S
E
F
INVERSER (V)
S
V
F
S
E
V
S
V
S
S

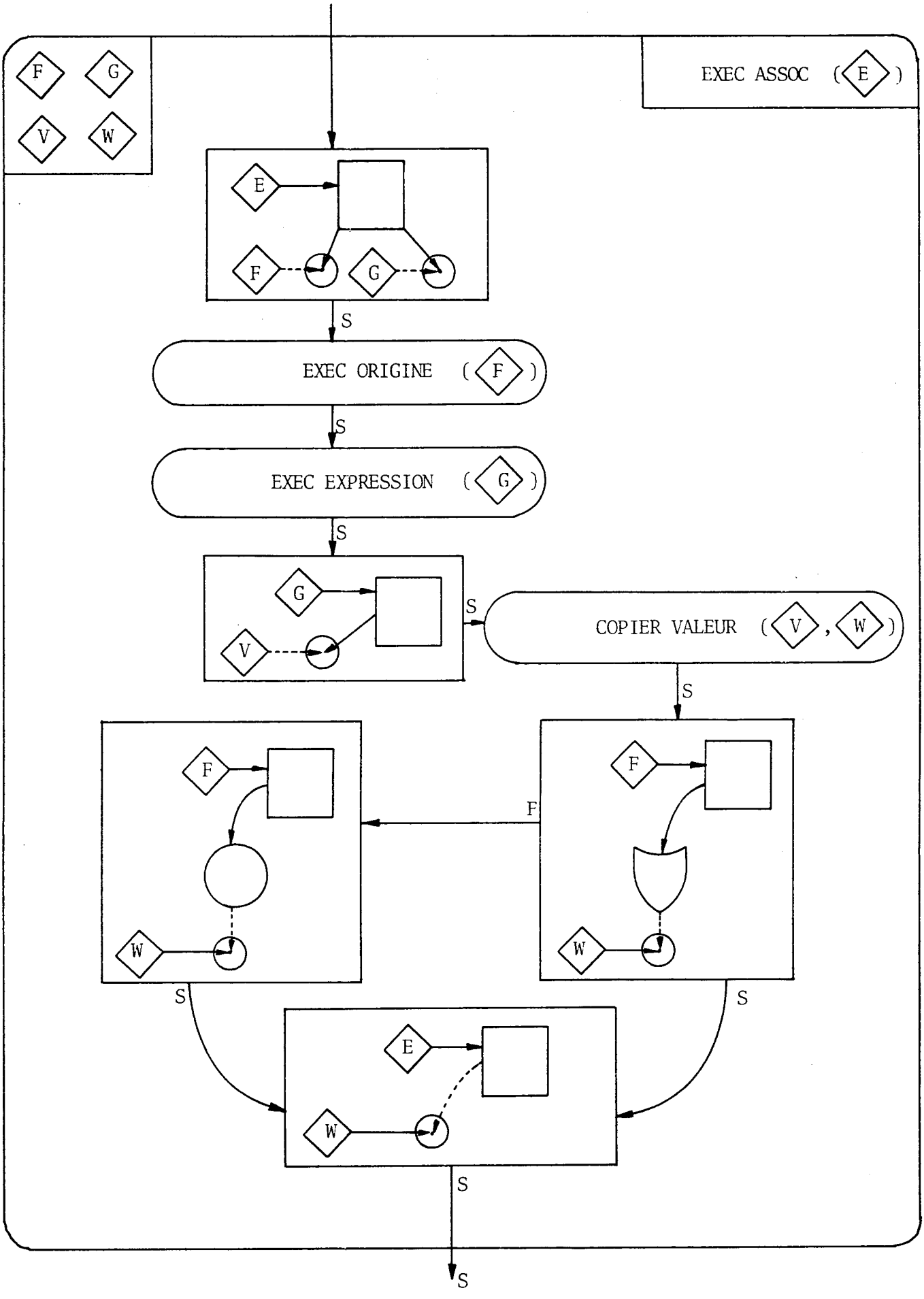

F
G
EXEC BLOC (E)
E
F
G
S
EXEC PROLOGUE (F)
F
S
S
EXEC BRANCHES (G)
S
E
G
S
S

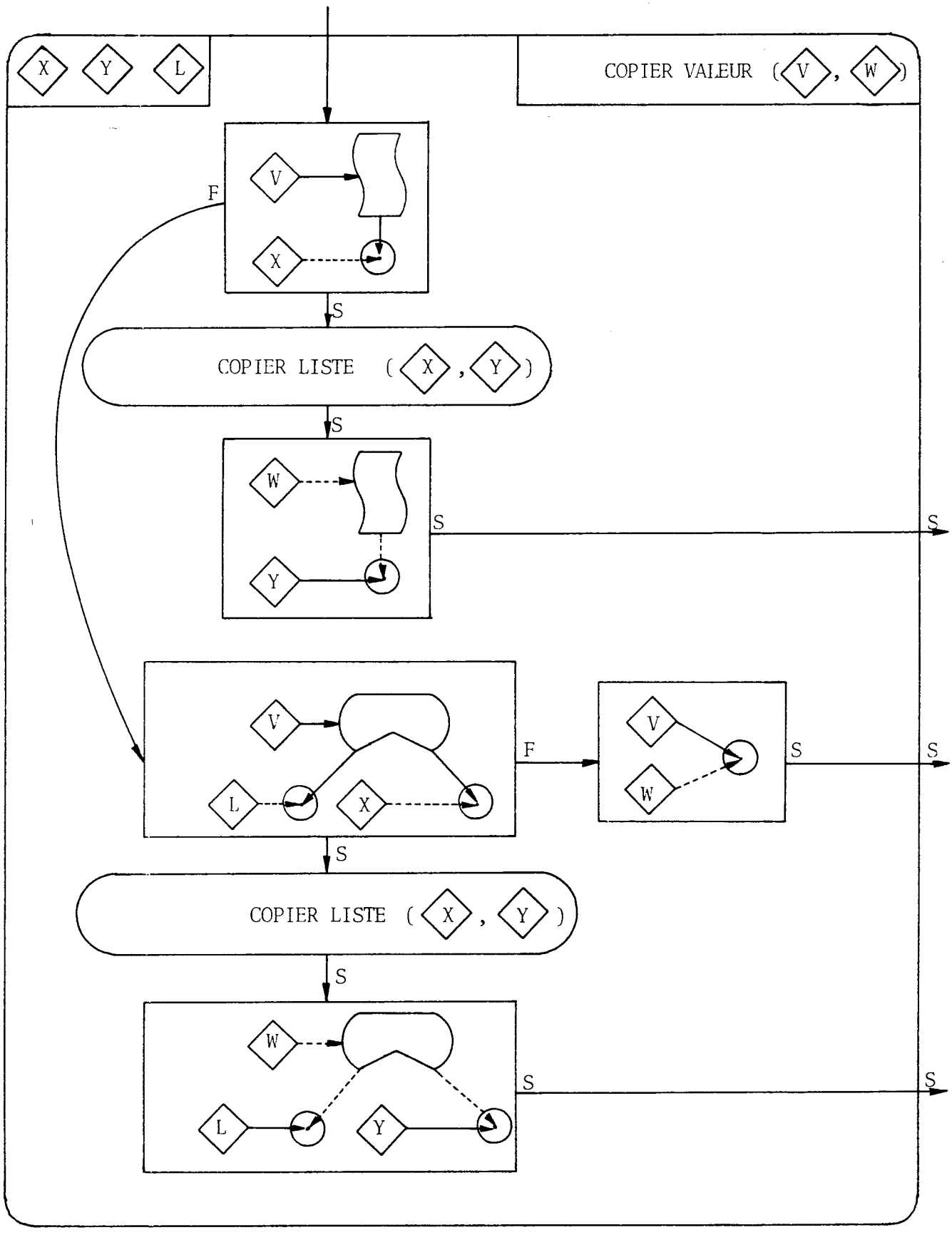

F
G
EXEC PROLOGUE (E)
E
F
G
S
F
S
G
S
S
F
F
G
F
F
EXEC CONDIT (F)
EXEC ACTIV (G)
S
S
F
G
S
S

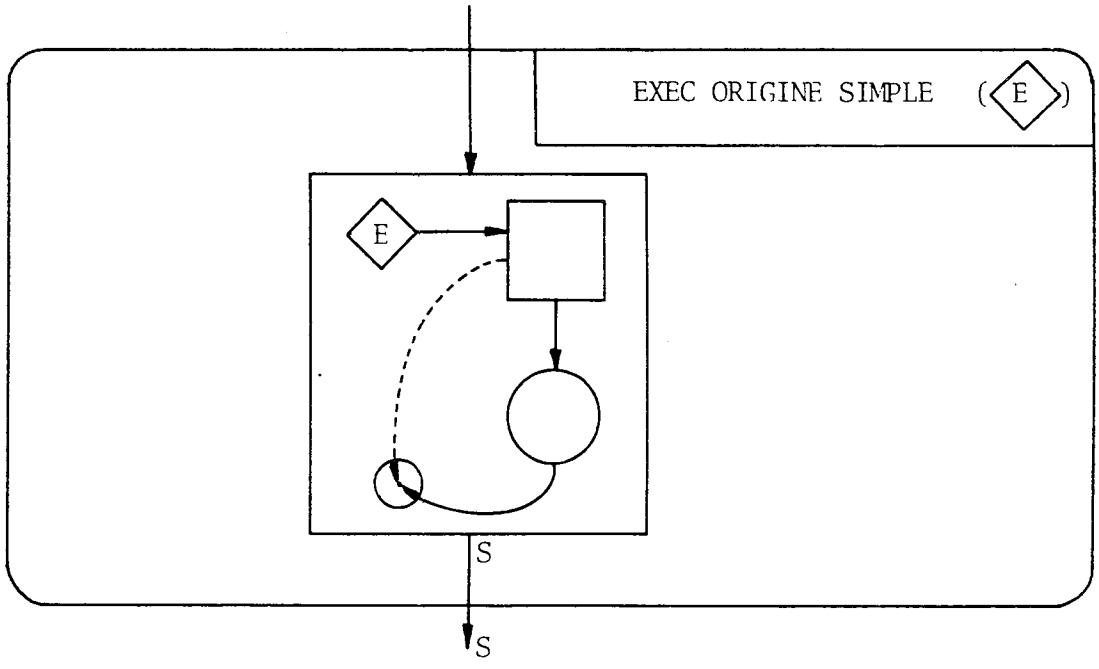

P
Q
EXEC CONDIT (E)
E
P
Q
S
MEME MODE (P , Q)
S
E
S
F
S
V
W
EXEC ACTIV (E)
E
V
S
COPIER VALEUR (V , W)
S
E
W
S
S

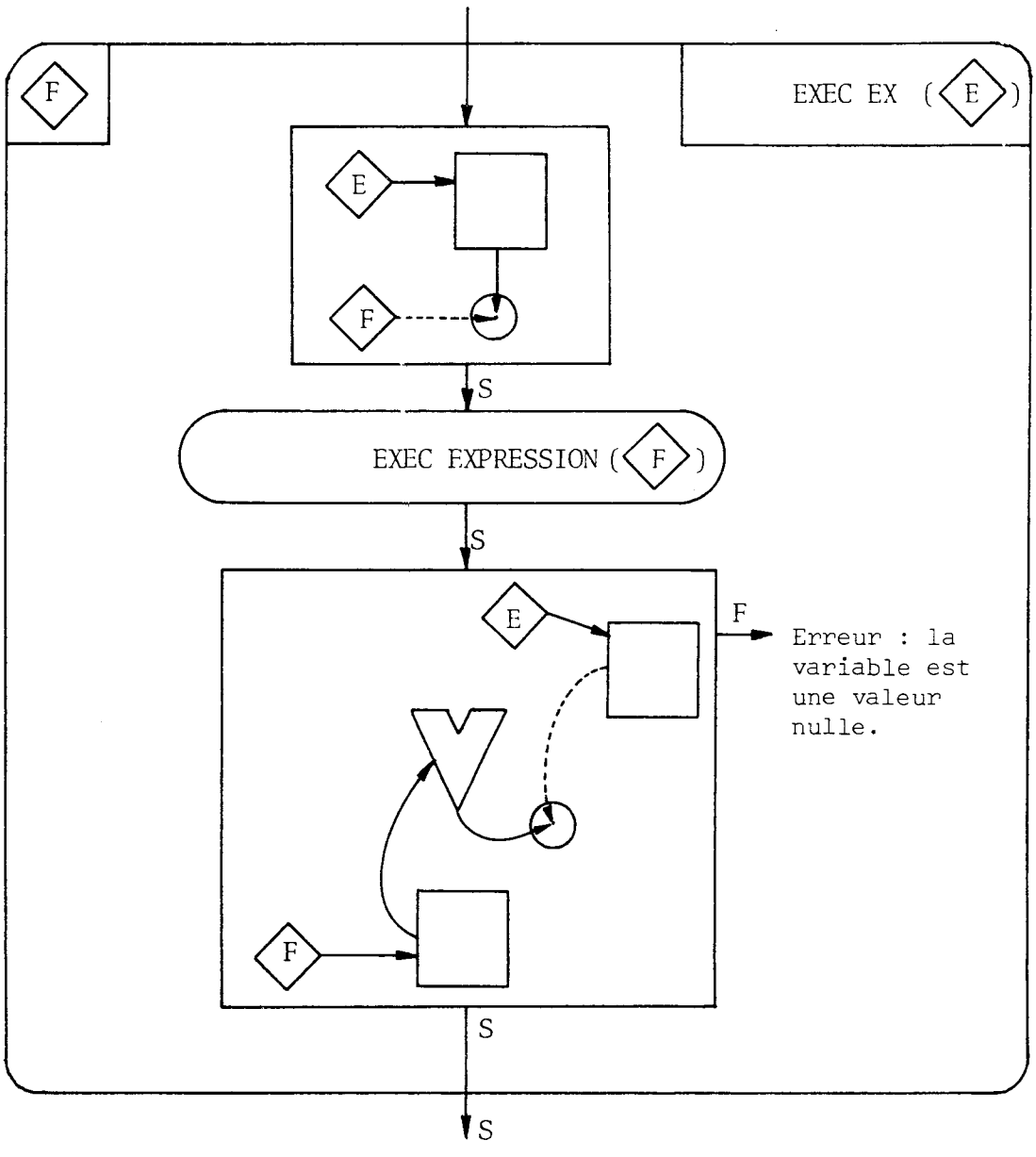

F
EXEC BRANCHES (E)
E
F
S
EXEC SEQUENCE (E , F)
S
F
S
E
F
S
S
F
F
V
S
S
F
S
F
E
S
F

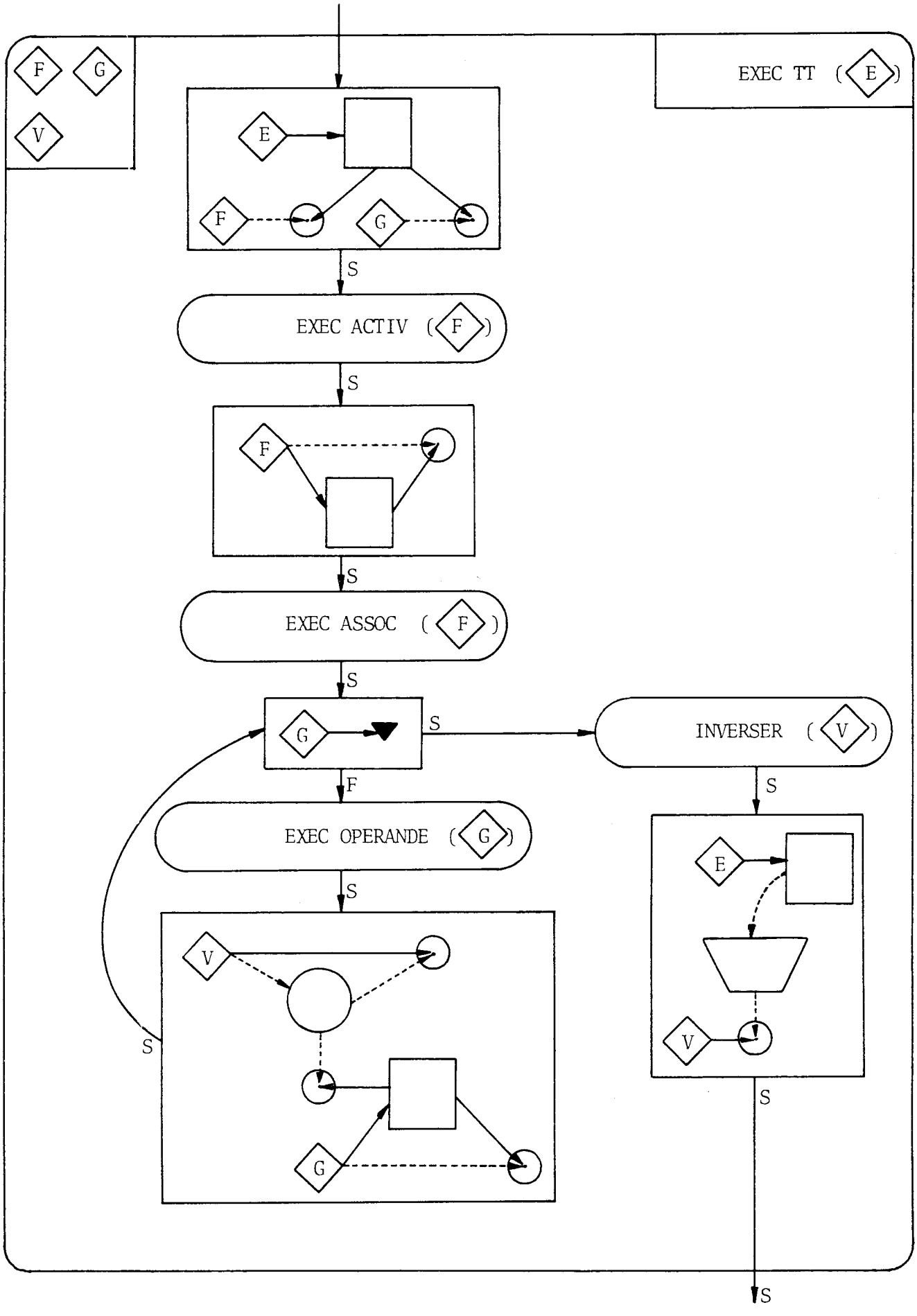

G
EXEC SEQUENCE (E , F)
F
G
S
allera
F
EXEC EXPRESSION (G)
G
S
G
S
F
E
F
G
F
G
S
S
EXEC EXPRESSION (E)
assoc.
E
F
EXEC TERME (E)
S
S
EXEC ASSOC (E)
S
S
S

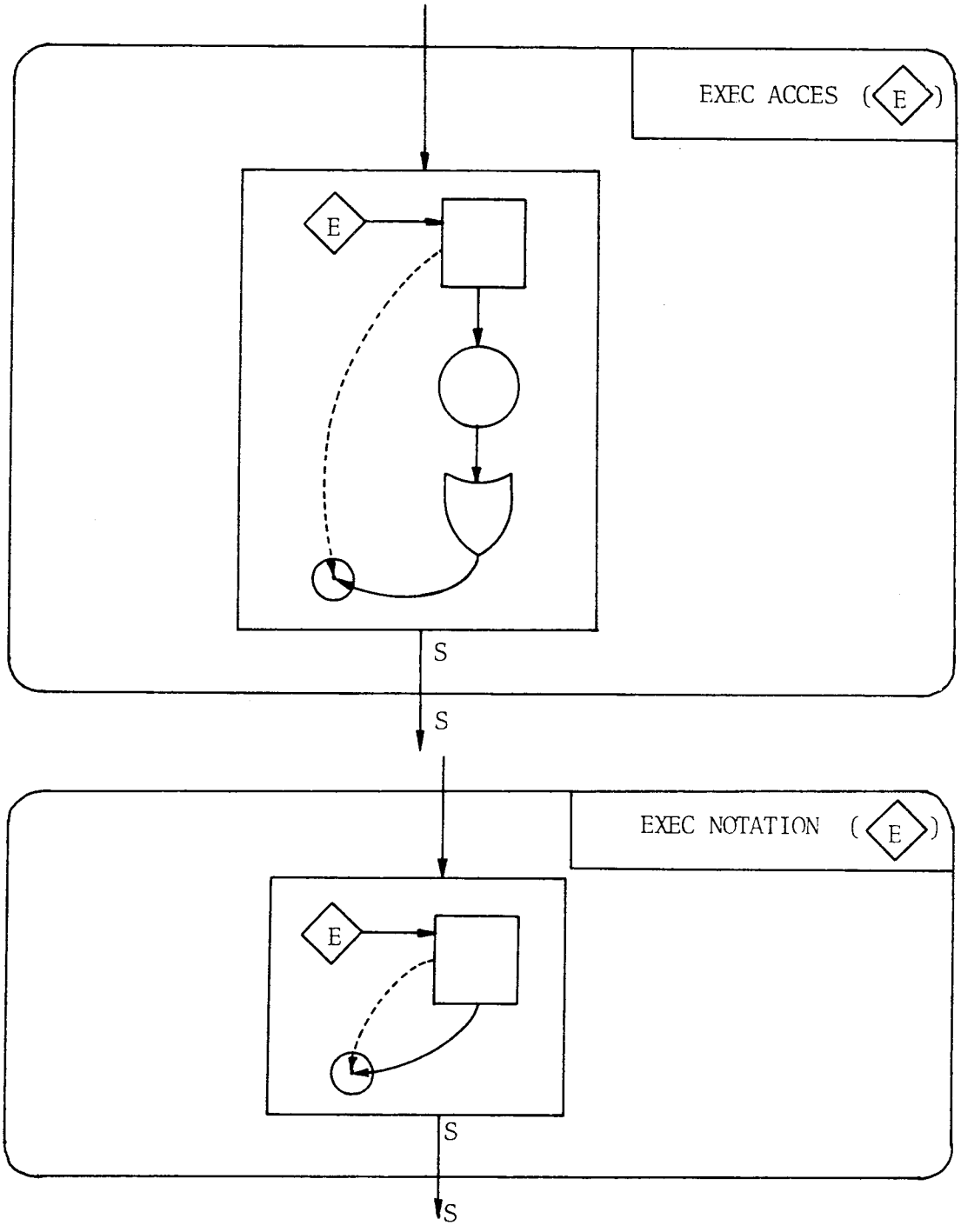

F
G
V
W
EXEC ASSOC (E)
E
F
G
S
EXEC ORIGINE (F)
S
EXEC EXPRESSION (G)
S
G
V
S
COPIER VALEUR (V , W)
S
F
W
F
F
W
S
S
E
W
S
S


X
Y
L
COPIER VALEUR (V , W)
V
X
F
S
COPIER LISTE (X , Y)
S
W
Y
S
S
V
L
X
F
V
W
S
S
S
COPIER LISTE (X , Y)
S
W
L
Y
S
S


EXEC ORIGINE SIMPLE (E)
E
S
S


F
EXEC EX (E)
E
F
S
EXEC EXPRESSION (F)
S
E
F
Erreur : la variable est une valeur nulle.
F
S
S


F
G
V
EXEC TT (E)
E
F
G
S
EXEC ACTIV (F)
S
F
S
EXEC ASSOC (F)
S
G
S
INVERSER (V)
F
S
EXEC OPERANDE (G)
S
E
V
V
G
S
S
S


EXEC ACCES (E)
E
S
S
EXEC NOTATION (E)
E
S
S

3.1.5. QUELQUES REMARQUES FINALES SUR BASEL ET SA DEFINITION

Au cours de cet exposé sur BASEL, on a pu voir que ce langage contient une variété de mécanismes qui dépassent de beaucoup les possibilités que l'on avait établies au départ comme étant celles du langage de base d'un langage extensible. En effet, comme on le verra plus en profondeur au début du chapitre suivant, BASEL contient déjà un embryon d'extensibilité sémantique avec son mécanisme de modes et ses procédures génériques, de la même façon qu'Algol 68 avec ses modes et ses opérateurs [vW74]. Par bien des aspects, BASEL ressemble en effet à Algol 68.

Cependant, sur un certain nombre de points importants, par la clarté et la simplicité de sa structure, on peut affirmer que BASEL propose des solutions mieux organisées et plus faciles à comprendre que leurs homologues en Algol 68 :

- une expression, quelle qu'elle soit, a toujours un mode qui lui est propre. Ce n'est pas le cas en Algol 68, où, par exemple, les expressions collatéralles n'ont pas de mode tant que le contexte ne leur en a pas imposé un, et où les listes de paramètres n'ont pas de mode du tout.
- La notion de tuple comme objet servant uniquement à rassembler d'autres objets destinés à un usage déterminé, rend systématique et uniforme le traitement de nombreux points qui autrement seraient autant de cas particuliers :
    - expressions collatéralles,
    - listes de paramètres, à évaluation sérielle ou collatéralle,
    - appels de procédures avec nombre de paramètres "variable",
    - formation de structures,
    - formation de séquences,
    - etc....
- La formation d'objets composés, par les tuples, peut se faire de façon sérielle ou de façon collatéralle, alors qu'en Algol 68 elle n'est possible que de façon collatéralle (il ne semble d'ailleurs pas y avoir de raison technique à cette restriction).

- L'itération permet de construire des séquences, alors qu'il n'existe pas, en ALGOL 68, d'expression permettant de construire des objets dont le nombre d'élément est déterminé dynamiquement.

- La notion de branche d'instructions rassemble sous une forme unique ce qui, en Algol 68, est réalisé d'une part par l'expression conditionnelle et d'autre part par les blocs avec exit.

- La déclaration conditionnelle, en portant au niveau du bloc le test dynamique de mode assure une protection totale de l'objet de mode union, sans nécessiter une copie de sa valeur, comme c'est le cas en Algol 68.

- Les problèmes de portée ne se posent pas en BASEL, sans que cela devienne une gène pour la programmation. En particulier, les procédures sont vraiment des objets "à part entière" alors que ce n'est pas le cas, à cause des problèmes de portée, en Algol 68.

D'autre part, la définition formelle de BASEL donne de ce langage une description utilisable très directement par l'implémenteur, bien qu'étant complètement indépendante d'une machine particulière. C'est, à mon avis, le rôle essentiel de la définition formelle d'un langage de programmation que de pouvoir être suivie aussi fidèlement que possible par l'implémenteur du langage, car il peut alors affirmer qu'il a bien implémenté le langage défini et non pas un quelconque dialecte accidentel. Une argumentation assez complète sur ce sujet a d'ailleurs été faite dans [Jo72a]. Pour BASEL, ceci a été vérifié lors de l'expérience faite aux Etats-Unis : 8 mois, avec 3 personnes, ont suffit pour réaliser une implémentation, expérimentale certes, mais complète. On ne peut pas en dire autant de l'utilisation que peuvent faire aujourd'hui du rapport Algol 68 les implémenteurs actuels de ce langage, toute proportion étant gardée quant à la complexité du langage.

Enfin, la syntaxe de BASEL a délibèrement été choisie de nature rudimentaire, car un mécanisme d'extension syntaxique faisait partie des composants envisagés pour ELF. Ainsi, BASEL n'a pas les opérateurs d'Algol 68, mais seulement leur dimension sémantique avec les procédures génériques.

En tant que langage de base d'un langage extensible, on peut cependant estimer que BASEL est trop "évolué". En effet, il prédéfinit un certain nombre d'aspects qui, comme on le verra dans le chapitre suivant, devraient plutôt être obtenus par extension si on désire les avoir dans le langage que l'on utilise. Parmi ceux-ci, les plus marquants sont les modes, leur mécanisme de construction et l'organisation du mécanisme des modifications. Ce sont là des points qui permettent de douter de la qualité de BASEL en tant que langage de base : il fournit des éléments qui ne sont pas assez rudimentaires, dans le domaine sémantique, pour qu'il soit possible de construire librement des langages étendus dans lesquels on désire peut être, par exemple, un mécanisme de modification tout à fait différent de celui de BASEL.

Dans le paragraphe qui suit, on esquisse quelques caractéristiques d'un langage qui pourrait constituer un langage de base remplissant son rôle de façon beaucoup plus stricte que ne le fait BASEL.

3.2. LO : AUTRE APPROCHE POUR UN LANGAGE DE BASE

Ce qui suit n'est qu'une ébauche. On n'y décrit pas un langage dans sa totalité, mais plutôt, avec plus ou moins de détails selon les cas, certains points importants qui font qu'un langage peut être pris comme langage de base d'un langage extensible en laissant le maximum de liberté quant à la forme et au contenu des langages étendus qui pourront être construits à partir de lui. On supposera bien sûr l'existence de mécanismes d'extension, aussi bien syntaxiques que sémantiques, mais sans en préciser le fonctionnement exact.

La plupart des remarques et des propositions qui suivent peuvent être rassemblées à l'intérieur d'un même langage, que l'on appellera L0, et dont ne seront donc décrits que les aspects qui sont intéressants ici. La plupart de ces remarques et de ces propositions sont issues d'un travail en commun avec D. Bert [Jo72], qui a été approfondi dans sa thèse [Be73] pour étudier surtout les problèmes relatifs au contrôle de l'exécution, et d'un cours de D.E.A. sur les langages extensibles que j'ai fait à l'Institut de Programmation de l'Université de Paris VI pendant les années 1972/73 et 1973/74.

3.2.1. STRUCTURE GENERALE DE LO

Le langage L0 permet de manipuler des objets que l'on peut ranger en deux catégories :

- les textes de programmes ;
- les valeurs et la mémoire.

3.2.1.1. Les textes de programmes

L'élément de base pour construire les programmes en L0 est un objet appelé module, qui est de la catégorie "texte de programme". Il y a deux façons d'obtenir un module :

- soit en écrivant une "notation" de module ;
- soit en spécifiant la façon dont ce module est construit par un "générateur de modules", appelé, en L0, un lambda. Un lambda est, lui aussi, un objet de la catégorie "texte de programme".

Modules et lambdas sont les deux seuls objets de cette catégorie.

A. Les modules

Syntaxiquement, une notation de module est définie par :

notation-de-module : module (corps-de-module)
corps-de-module : [environnement,] corps-de-module-simple
environnement : env (identificateur <, identificateur> ...)
corps-de-module-simple : declarations , expression |
declarations |
expression
declarations : declaration <, declaration> ...
declaration : def (identificateur , expression)

A.1. Les déclarations

Toutes les déclarations ont la forme :

def (I, E)

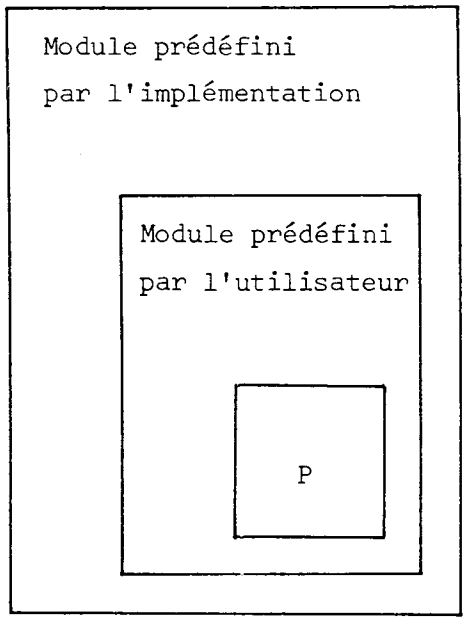
où I est un identificateur et E une expression.

Une déclaration définit l'identificateur I dans le module où elle est écrite comme étant synonyme de l'objet obtenu en résultat de l'expression E au moment où ce module est évalué (voir le paragraphe C ci-dessous).

Une expression écrite dans un module pouvant elle-même être un notation de module, on obtient ainsi une imbrication similaire à celle qui existe dans les langages à structure de blocs, et qui obéit à des règles analogues sur la portée des déclarations.

A.2. L'environnement

En L0, le programme est un module et son organisation rappelle celle des programmes Algol 68, avec leurs "prologues" et leurs "programmes particuliers". En effet, en L0, tout programme P écrit par un utilisateur est placé à l'intérieur de deux autres modules prédifinis, l'un par l'implémentation, l'autre part l'utilisateur lui-même. On peut schématiser ceci de la façon suivante :



Donc, à l'intérieur de P, selon les règles classiques de portée des déclarations, il est possible d'accéder à ce qui a été déclaré dans les deux modules prédéfinis. D'autre part, un module étant lui-même un objet, il peut être le résultat de l'expression E écrite dans une déclaration :

def (M, E)

Par exemple, E peut être une notation de module ou une expression qui génère un module à partir d'un lambda.

Une telle déclaration de module peut apparaître n'importe où, dans le module prédéfini de l'implémentation, dans le module prédifini de l'utilisateur ou dans le module P lui-même qui constitue le "programme particulier".

Soit alors N une notation de module. N a la forme :

module (C)

où C est un corps de module.

L'endroit où cette notation N est écrite est dans la portée d'un certain nombre de déclarations de modules de la forme :

def (M, E)

Soient, choisis parmi ces déclarations, M1, M2, ..., Mn, n identificateurs déclarés comme étant synonymes de modules, dont les notations respectives peuvent être écrites sous la forme :

module (D11, D12, ..., D1p1)

module (D21, D22, ..., D2p2)

...

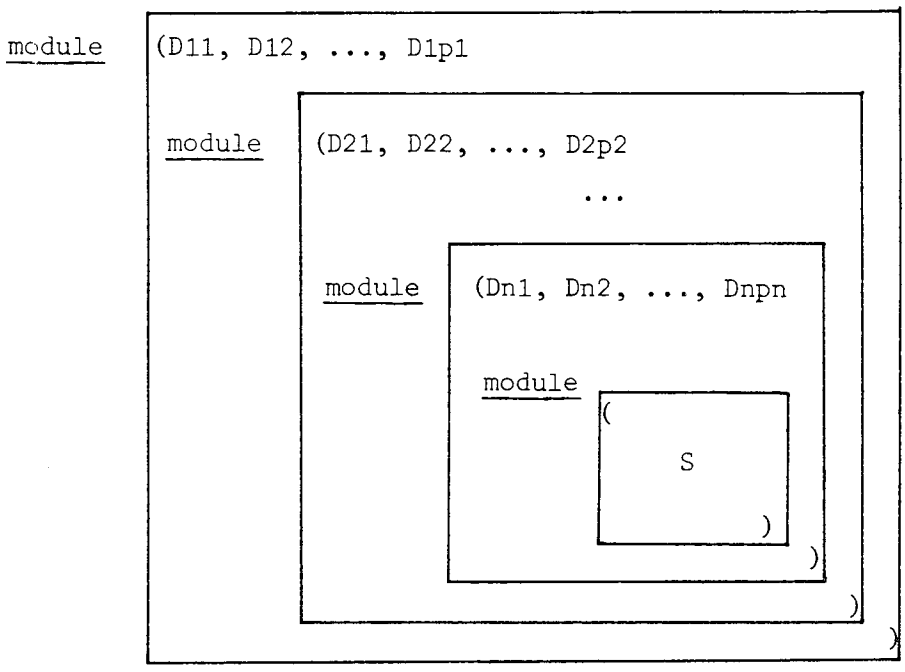
module (Dn1, Dn2, ..., Dnpn)

où les Dij sont des déclarations.

Finalement, si le contenu C de la notation N est de la forme :

env (M1, M2, ..., Mn), S

où S est un corps de module simple, ceci est équivalent à remplacer N par l'imbrication :



On dit alors que env(M1, M2, ..., Mn) établit un environnement pour S, à partir des modules prédéfinis M1, M2, ..., Mn.

Ce mécanisme de construction d'environnement à partir de modules prédéfinis va dans le sens des travaux les plus récents sur la programmation modulaire.

En particulier, c'est à partir d'idées de ce genre que Schuman à conçu un système de programmation modulaire pour langage de haut-niveau. Une description initiale de ce système est donnée en [Sc74a] pour un langage du genre d'Algol 68. Cependant, l'intention n'est pas d'introduire dans L0 les mécanismes très variés dont Schuman a muni son système de programmation modulaire : seul est gardé ici le principe de base de la construction de modules indépendants et celui de leur composition pour constituer des environnements.

B. Les lambdas

Un lambda est une expression avec paramètres formels, et la liaison des paramètres d'un lambda avec des paramètres effectifs génère un module.
La forme d'une notation de lambda est définie par :

notation-de-lambda : lambda (identificateur, <identificateur,>... expression)

où la liste d'identificateurs définit les paramètres formels.

Comme pour les modules, un lambda étant un objet, il peut être le résultat de l'expression E écrite dans une déclaration :

def (L, E)

Si K est un lambda, la génération d'un module par K est une expression de liaison qui a la forme :

lier (K, E1, E2, ..., Ep)

où les Ei sont des expressions qui constituent les paramètres effectifs.
Si K a la forme :

lambda (I1, I2, ..., Ip, E)

le module généré est :

module (D1, D2, ..., Dp, E)

où Dj est de la forme :

def (Ij, Ej)

On peut ainsi, avec des paramètres effectifs différents générer un nombre quelconque de modules à partir d'un même lambda.

C. L'évaluation

Soit un module M de la forme :

module (D1, D2, ..., Dp, E)

où les Di sont des déclarations et E une expression.
Ce module, qu'il soit obtenu directement par notation, ou qu'il soit généré par liaison des paramètres d'un lambda, est un texte de programme dont le rôle est de décrire, avec son expression E, un certain nombre d'actions qui doivent se dérouler dans l'environnement défini par ses déclarations et par son imbrication statique dans le programme.

Ces actions sont déclenchées, dynamiquement, par l'exécution d'une expression d'évaluation, qui a la forme :

eval (M)

Si M est une notation de module, ceci est donc écrit :

eval (module (D1, D2, ..., Dp, E))

et il s'agit de l'analogue d'un bloc dans un langage de la famille Algol.
Si M est simplement un identificateur déclaré comme étant synonyme d'un module, eval (M) est alors l'analogue d'un appel de procédure sans paramètre.

Si M est une expression de liaison, l'évaluation a la forme :

eval (lier (L, E1, E2, ..., Ep))

et il s'agit de l'analogue d'un appel de procédure avec paramètres.

Dynamiquement, l'évaluation d'un module se déroule de façon classique.
En utilisant des termes empruntés à la description de BASEL (paragraphe 3.1.2., section C), il y a trois phases dans cette évaluation :

- le prologue, qui crée l'activation actuelle à partir des déclarations ;
- le corps de l'évaluation, qui exécute E ;
- l'épilogue, qui détruit l'activation créée dans le prologue et prend pour activation actuelle l'activation qui lui était immédiatement antérieure.

Pendant le prologue, une activation est créée à partir de l'ensemble des déclarations du module, sans aucune spécification de l'ordre dans lequel chaque déclaration est prise en compte. Une déclaration de la forme :

def (I, E)

commence par évaluer E et elle associe à I l'objet obtenu au résultat : soit un "texte de programme", soit une "valeur" ou un "emplacement" dans la mémoire.

Par exemple :

def (M, module (...))

associe à M le module dont la notation est écrite dans la déclaration, tandis que :

def (V, eval (module (...)))

associe à V le résultat de l'évaluation du module dont la notation est écrite dans la déclaration.

D'autre part, la destruction d'une activation, effectuée en épilogue, ne fait que détruire la relation d'association identificateur-objet établie par chaque déclaration, mais elle ne détruit pas les objets eux-mêmes.

Enfin, le résultat de l'évaluation d'un module est le résultat de son expression E.

Par ailleurs, si un module M a la forme :

module (env (M1, M2, ..., Mn), D1, D2, ..., Dp, E)

l'évaluation eval (M) a pour effet :

```
eval (module (D11, D12, ..., D1p1,
     eval (module (D21, D22, ..., D2p2,
                   ...
                eval (module (D1, D2, ..., Dp, E)) ...))))
```

où Dk1, Dk2, ..., Dkpk sont les déclarations qui constituent tout le corps du module Mk ($1 \leq k \leq n$).

L'évaluation des modules sans déclarations est un cas particulier trivial de ce qui vient d'être dit et celle des modules sans expression n'a qu'un intérêt fort limité, le rôle de tels modules étant surtout de permettre la construction d'environnements pour d'autres modules.

3.2.1.2. Les valeurs et la mémoire

LO permet l'utilisation de valeurs de diverses sortes et la détermination d'emplacements dans un modèle de la mémoire pour ranger ces valeurs et organiser des ensembles de valeurs. Cet aspect très primitif de LO correspond au souci de laisser la liberté de choisir la représentation interne des objets sur des machines dont l'architecture est celle que l'on connait aujourd'hui. Pour des machines ayant une organisation interne différente - qui ne laisserait pas transparaître, par exemple, la structure linéaire de la mémoire - il faudrait reconsidérer cet aspect de LO.

A. Les valeurs

Quelques points généraux suffiront à indiquer le "style" de valeurs que LO permet d'utiliser :

- le choix des sortes de valeurs primitives est fait dans le souci de rendre aussi directe que possible leur représentation sur une machine.
- La notion de type n'existe pas, les diverses sortes de valeurs se distinguant entre-elles uniquement par la forme de leur représentation interne : virgule fixe, virgule flottante, suites de bits de longueur quelconque, etc.... Ce sont les opérations qui utilisent les valeurs qui les considérent, implicitement, comme ayant une représentation donnée.
- A chaque sorte de valeur correspond une notation ou, au moins, une opération primitive de LO pour passer d'une suite de marques typographiques à une représentation interne. Par exemple :

  123 est mis sous forme "virgule fixe"
  3.14 est mis sous forme "virgule flottante"
  bit (1011101101) construit une suite de bits.

Ainsi, on peut déclarer :

def (PI, 3.14159)

Au moment où est évalué le module qui contient cette déclaration, PI est rendu synonyme de la représentation en virgule flottante de 3.14159.

B. La mémoire

D'autre part, toujours dans le souci de donner à LO son rôle de langage de base, c'est-à-dire, en particulier, d'intermédiaire entre des langages étendus quelconques et une machine, LO propose aux mécanismes d'extension une vue très rudimentaire de la mémoire : une simple suite d'"unités de mémoire", qui peuvent d'ailleurs déjà être organisées en multiples les unes des autres, comme par exemple les octets, demi-mots, mots et double mots de certaines machines. Ce modèle de la mémoire est rendu accessible à travers un répertoire d'opérations. Ainsi, dans le cas d'une mémoire organisée simplement en mots, on aurait l'opération :

allouer (n)

pour créer un emplacement constitué de n mots contigüs, le résultat de cette opération étant un nom interne pour cet emplacement.

Une autre opération permet aussi de déterminer un emplacement à l'intérieur d'un autre emplacement, en donnant sa taille et le déplacement de son origine par rapport à l'origine de l'emplacement contenant.Etc. Ainsi, de la même façon que les valeurs disponibles en LO permettent la représentation d'objets primitifs, le modèle de la mémoire permet de représenter l'organisation et la structure d'un ensemble de valeurs. Un certain nombre de propositions dans ce sens ont été faites dans [Jo72] et [Be73].

C. Opérations sur les valeurs et la mémoire

Pendant l'évaluation d'un module, l'association identificateur-objet est établie lors du prologue et détruite lors de l'épilogue. Aucune opération n'a la possibilité d'altérer cette association pendant le corps de l'évaluation. Ainsi, les déclarations créent des constantes, à la manière des déclarations d'identité qui existent en Algol 68. De même, comme les repères en Algol 68, ce sont les emplacements qui apportent en LO la notion classique de variable.

Si E est une expression qui a pour résultat une valeur V et si F est une expression qui a pour résultat un emplacement A, l'expression :

copier (E, F)

range dans A une copie de V, le résultat de cette expression étant A, représenté par son nom interne.

D'autre part, si F est une expression qui a pour résultat un emplacement A, l'expression :

val (F)

a pour résultat une copie du contenu de A.

Ainsi, une affectation classique du genre X := Y, au niveau primitif du langage de base L0 est représentée par :

copier (val (Y), X)

Sur les valeurs, L0 permet d'effectuer un certain nombre d'opérations, comme :

-arithmétique en virgule fixe : addfix (E, F)
sousfix (E, F)
prodfix (E, F)
divfix (E, F)
mod (E, F)
absfix (E)
signfix (E)

- arithmétique en virgule flottante : addflott (E, F)
sousflott (E, F)
etc...

- opérations de comparaison : supfix (E, F)
egafix (E, F)
etc ...
supflott (E, F)
egflott (E, F)
etc...

- transformations de représentations : fixflott (E)
  flottfix (E)

- opérations sur les suites de bits : ou (E, F)
  exou (E, F)
  et (E, F)
  non (E)

Cependant, ce qu'il faut surtout remarquer ici, ce n'est pas d'abord la liste de ces opérations, mais le fait qu'elles opèrent sur des représentations internes de valeurs, en ignorant complètement toute notion de type. En effet, L0 est un langage sans type, ce qui signifie avant tout qu'aucun contrôle n'est effectué par ses opérations sur la nature des opérandes qui leur sont fournis : elles opèrent toutes en supposant que leurs opérandes sont sous la forme d'une représentation donnée. Par exemple, l'addition en virgule fixe prend ses deux opérandes tels qu'ils sont, et travaille avec eux en les considérant comme des nombres représentés en virgule fixe. C'est cet aspect de L0 qui en fait vraiment un langage sans type.

3.2.1.3. Expressions composées

Les identificateurs, les notations de valeurs, les opérations sur les valeurs et la mémoire, les liaisons de paramètres de lambdas, les évaluations de modules et les notations de modules et de lambdas sont diverses formes d'expressions simples qui peuvent être écrites dans un module ou dans un lambda.

Il existe également des expressions composées qui permettent de regrouper et de structurer des ensembles d'expressions. Pour plusieurs d'entre elles, on reconnaitra d'ailleurs des constructions qui existaient déjà en LISP 1.5 [Mc62].

A. La série d'expressions

Soient E1, E2, ..., En des expressions.
L'expression :

serie (E1, E2, ..., En)

signifie l'évaluation sérielle de son contenu : E1, puis E2, ..., puis En.
Le résultat d'une série est le résultat de En.

B. Les conditions

Comme en Algol 68, il y a deux sortes de conditions :

- la condition booléenne,
- la condition entière.

B.1. La condition booléenne

Soient C1, C2, ..., Cn, E1, E2, ..., En, En+1 des expressions.
L'expression :

cond ((C1, E1), (C2, E2), ..., (Cn, En), En+1)

demande l'évaluation des Ci les uns après les autres, dans l'ordre où ils sont écrits. Dès que le résultat de l'un d'eux est la valeur correspondant à la notation bit(1) - représentation conventionnelle de la notion "vrai" - l'expression Ei correspondante est évaluée et son résultat est celui de la condition. Si aucun Ci n'est "vrai", c'est En+1 qui est évalué et c'est son résultat qui est pris comme valeur de la condition.

B.2. La condition entière

Soient C, E1, E2, ..., En, En+1 des expressions.
L'expression :

cas (C, E1, E2, ..., En, En+1)

demande d'abord l'évaluation de C. son résultat est pris comme étant un nombre en virgule fixe. Si l'entier i correspondant est tel que $1 \leq i \leq n$, Ei est évalué et son résultat devient résultat de la condition. Si $i < 1$ ou $i > n$, c'est En+1 qui est évalué et son résultat devient résultat de la condition.

C. Les itérations

Il y a deux sortes d'itérations :

- le "tantque" ;
- le "jusqu'à".

Soient E1, E2 et E3 des expressions.

C.1. Le "tantque"

L'expression :

tantque (E1, E2, E3)

évalue d'abord E1. Si sa valeur est "vrai" (bit (1)), alors E2 est évalué. Puis E1 est à nouveau évalué, etc.... Tantque E1 est "vrai", E2 est évalué. Dès que E1 n'est plus "vrai", c'est E3 qui est évalué et l'évaluation du "tantque" est alors terminée. Son résultat est celui de E3.

C.2. Le "jusqu'à"

L'expression :

jusqua (E1, E2, E3)

évalue d'abord E1. Puis elle évalue E2. Si E2 n'est pas "vrai", E1 est à nouveau évalué, puis E2. Si E2 n'est toujours pas "vrai", E1 est encore évalué, etc.... Dès que E2 devient "vrai", E3 est évalué. L'évaluation du "jusqu'à" est alors terminée et son résultat est celui de E3.

3.2.2. REMARQUES SUR LO

Ce rapide survol de quelques points importants du langage que l'on a baptisé LO appelle quelques commentaires sur la qualité d'un langage de ce genre comme langage de base d'un langage extensible.

A - Pouvoir considérer que certaines constructions syntaxiques, que l'on a appelées "textes de programmes" sont des objets à part entière est important dans un langage extensible. En effet, les mécanismes d'extension ont essentiellement pour rôle de reconstruire, à l'aide du langage de base, ce qui est exprimé dans un langage étendu. Avoir dans le langage de base des objets qui facilitent cette construction est certainement souhaitable.

B - L'absence de la notion de type dans le langage de base n'implique pas que les langages étendus seront, eux aussi, sans types, c'est-à-dire sans le contrôle, statique ou dynamique, qu'implique en général cette notion. D'ailleurs, le mécanisme présenté au paragraphe 4.4. permet de construire complètement un système de types, avec types de bases, types construits, contrôle statique et dynamique, conversions automatiques, et il est présenté en prenant LO comme langage de base.

Ainsi, le langage de base n'imposant rien quant aux types de base ni aux contrôles à effectuer, liberté totale est laissée de définir ce que l'on veut dans les langages étendus, à condition, bien sûr, d'avoir un mécanisme d'extension le permettant. Cela n'est pas possible avec un langage de base comme BASEL qui, au contraire, imposerait forcément son "style" de modes et de modifications à tous les langages étendus définis à partir de lui.

C - La syntaxe de LØ est encore plus rudimentaire que celle de BASEL : elle est entièrement sous forme préfixée et parenthésée. Ceci ne doit pas être considéré comme une gène. En effet, il faut penser que l'utilisation normale doit être celle d'un langage étendu, l'usage du langage de base n'étant qu'exceptionnel et limité, en principe, aux portions de programmes (des modules prédéfinis) où sont utilisés les mécanismes d'extension pour, par déclarations appropriées, définir des langages étendus. C'est pourquoi il a été jugé préférable de rendre la forme du langage de base plus agréable aux mécanismes d'extension - syntaxiques en particulier - qu'au lecteur humain. D'ailleurs, la personne à qui est confiée la tâche de construire la définition d'un langage étendu n'est pas, en général, celle à qui est destiné ce langage.

D - Enfin, il faut considérer qu'un langage de base est bien plus un répertoire d'opérations qu'un langage complet et fermé du genre des Algols ou de BASEL. C'est d'ailleurs ce qui est suggéré par Schuman dans [Sc71] avec le concept d'interpréteur extensible, dont l'utilisation semble être un moyen de réaliser pratiquement l'unification entre définition formelle et implémentation, comme cela a été introduit dans [Jo72a]. De façon très schématique, l'interpréteur extensible permet de considérer que la sémantique du langage de base n'est pas "figée", et qu'une nouvelle opération peut toujours lui être rajoutée. Il s'agit là d'un travail qui doit être poursuivi car cela est certainement la façon d'envisager dans la pratique le noyau du fonctionnement d'un langage extensible réellement utilisable.

4. L'EXTENSION SEMANTIQUE

-:-:-:-:-:-:-:-:-:-:-:-:-:-

La sémantique d'un langage concerne les sortes d'objets manipulés par ce langage et les opérations qui peuvent être effectuées avec ces objets. Un mécanisme d'extension sémantique a donc pour rôle de permettre la définition de nouvelles sortes d'objets et des opérations associées.

Je présente ici trois mécanismes d'extension sémantique, tous construits sur un même principe général : la définition de types. Le premier mécanisme permet de réaliser des extensions sémantiques "superficielles" : c'est le mécanisme des modes classiques, qui ne remet pas en question la structure de la définition du langage, et qui s'adresse donc à l'utilisateur définissant lui-même, dans le langage, les sortes d'objets et les opérations qu'il veut utiliser.

Le deuxième mécanisme, celui des propriétés additionnelles, va un peu plus en profondeur dans la structure de la définition, car il permet de construire, en plus des modes classiques, des relations entre ces modes sous la forme de conversions, de prédicats et de fonctions diverses.

Quant au troisième mécanisme, appelé mécanisme des classes, il concerne toute la définition formelle de la sémantique statique d'un langage. Son principe de base et une introduction générale aux diverses possibilités envisageables de ce mécanisme sont présentés ici. Il est actuellement prévu l'étude détaillée d'un formalisme pour mettre en oeuvre ce mécanisme, ainsi que son utilisation pour la définition complète d'un langage de programmation : tout ceci doit faire l'objet de travaux ultérieurs, dont la direction générale est indiquée en [Sc74].

4.1. LA DEFINITION DE TYPES COMME MOYEN D'EXTENSION SEMANTIQUE

Dans un langage de programmation, un type définit, implicitement ou explicitement, les propriétés d'un certain ensemble d'objets. On peut distinguer deux catégories de propriétés ainsi définies par un type t :

- la façon dont sont construits les objets de type t ;
- les règles à respecter pour utiliser les objets de type t.

Ainsi, si l'on prend comme exemple le type entier d'Algol 60, aux objets de ce type sont associées un certain nombre de conventions (notation d'entier, virgule fixe, etc...) qui définissent la façon dont ils sont construits, et un certain nombre de règles sur la façon dont on peut les utiliser (addition de deux entiers, conversion d'un entier en réel, utilisation d'un entier comme indice, etc...).

De même en Algol 68, le mode struct(ent a, bool b) est celui d'objets qui ont la forme d'une suite de deux éléments, dont le premier est de mode ent et le second de mode bool, obtenue grâce à certains mécanismes spécialisés de construction, et qui peuvent être utilisés par un certain nombre d'opérations générales comme, par exemple, la sélection de l'un de ces éléments.

D'une façon générale, j'appellerai configuration statique des objets d'un certain type t tout ce qui concerne leur construction, et comportement dynamique tout ce qui définit les règles de leur utilisation.

La configuration statique des objets de type t définit donc :

- le type, l'ordre, et éventuellement le nombre des éléments qui constituent cet objet ;
- le mécanisme de construction de ces objets et, éventuellement, les notations qui lui correspondent.

Le comportement dynamique des objets de type t définit :

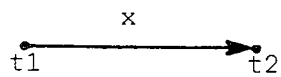
- les opérations du langage dont ces objets peuvent être les opérandes ;
- les conversions automatiques que ces objets peuvent subir pour être transformés en objets d'un autre type ;

- les conditions, ou prédicats, que ces objets doivent éventuellement satisfaire pour bien être des objets de type t, et pas seulement des objets d'un autre type t' ;
- et, bien sûr, l'ensemble des procédures ou fonctions de toutes sortes auxquelles ces objets peuvent être transmis comme paramètres.

On rejoint donc par là tout le domaine de la sémantique, car un type définit une catégorie d'objets et les opérations qui s'appliquent à ces objets. C'est pourquoi, dans ce qui suit, j'aborde l'extension sémantique à travers la défnition de types, appelés "modes" ou "classes" selon les mécanismes présentés.

Tout au long de la présentation de ces mécanismes, j'utilise une méthode uniforme pour décrire ce qu'ils permettent de définir : ce support unique de description est le graphe des propriétés dont j'ai donné une première idée dans [Sc70], que j'ai exploité un peu plus en profondeur dans [Jo71] et [Jo72] et qui a atteint sa forme actuelle, mais certainement non définitive, dans [Jo74]. C'est un graphe orienté dont le principe général de construction est le suivant : chaque sommet de ce graphe représente un type et chaque arc représente une propriété (configuration statique ou comportement dynamique) pour les objets du type représenté par le sommet situé à l'origine de cet arc.

Ainsi, dans :



l'arc peut représenter par exemple une conversion qui transforme un objet de type t1 en un objet de type t2, un prédicat qui, s'il est vérifié pour un objet donné de type t1, permet de considérer également cet objet comme un objet de type t2, ou encore une fonction d'une autre sorte. La nature et la description de la propriété représentée par l'arc sera toujours spécifiée dans l'information x portée par cet arc.

Toute extension sémantique sera alors reflétée par une modification dans le graphe : introduction d'un nouveau sommet ou construction d'un nouvel arc.

C'est l'utilisation de ce support unique de description qui va permettre de comparer de façon claire les mécanismes présentés.

4.2. LES MODES CLASSIQUES

Le principe de base des modes classiques, dont on trouve l'origine dans les travaux de Hoare [Ho66] et dans la thèse de Standish [St67], est le suivant :

- Un certain nombre de modes sont choisis comme étant des modes de base, le critère de choix étant surtout de faire correspondre ces types aux genres d'objets que sait directement traiter une machine.
- Un ensemble de constructeurs est établi, chaque constructeur pouvant être vu comme une "fonction" spécialisée qui prend comme arguments un certain nombre de modes - de base ou construits -, éventuellement des valeurs ou d'autres sortes d'objets, et qui retourne comme "résultat" un mode construit.
- Un système de déclaration permet de définir pour tout mode - construit ou non - un symbole qui sert à l'identifier.

BASEL, bien qu'étant un langage de base, utilise un tel mécanisme. Il en est de même pour Algol 68, bien que ce langage n'ait jamais été conçu pour être un langage extensible, et pour PPL, le langage extensible issu d'APL développé par Standish à la suite de sa thèse [St69, St71].

Ainsi, en Algol 68, les modes de base sont :

ent, reel, bool, car

Les constructeurs sont :

rep, qui prend un mode comme paramètre.
struct, qui prend une liste de modes et d'identificateurs comme paramètres.
proc, qui prend une liste de modes comme paramètres.
[ ], qui prend une liste de paires d'entiers et un mode comme paramètres.
union, qui prend une liste de modes comme paramètres.

Un mode construit est alors, par exemple :

struct (ent a, rep bool b)

et des déclarations de modes :

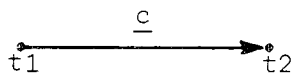
mode m1 = struct (ent a, rep bool b)
mode m2 = [1:N] proc m1

Soit donc m un mode, dans un langage du genre d'Algol 68 ou de BASEL.

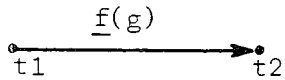
Si m est un mode de base, toutes les propriétés caractéristiques des objets de mode m sont prédéfinies :

- leur configuration statique est déterminée par des notations imposées dans le langage et par une représentation interne inaccessible ;
- leur comportement dynamique est déterminé par le jeu des conversions entre modes de base et par un répertoire figé d'opérations primitives.

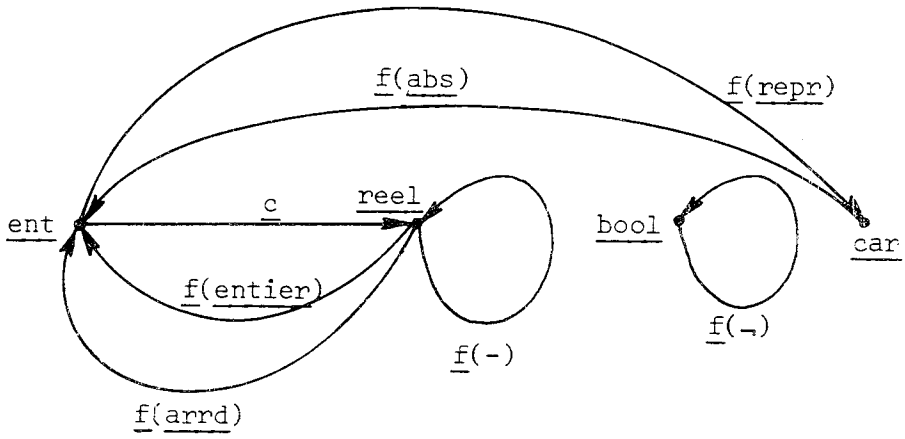
Donc, si on représente par :



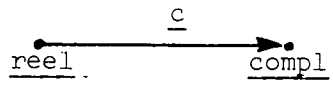
une conversion automatique qui prend une valeur de type t1 comme argument pour le transforemer en une valeur de type t2, et par :



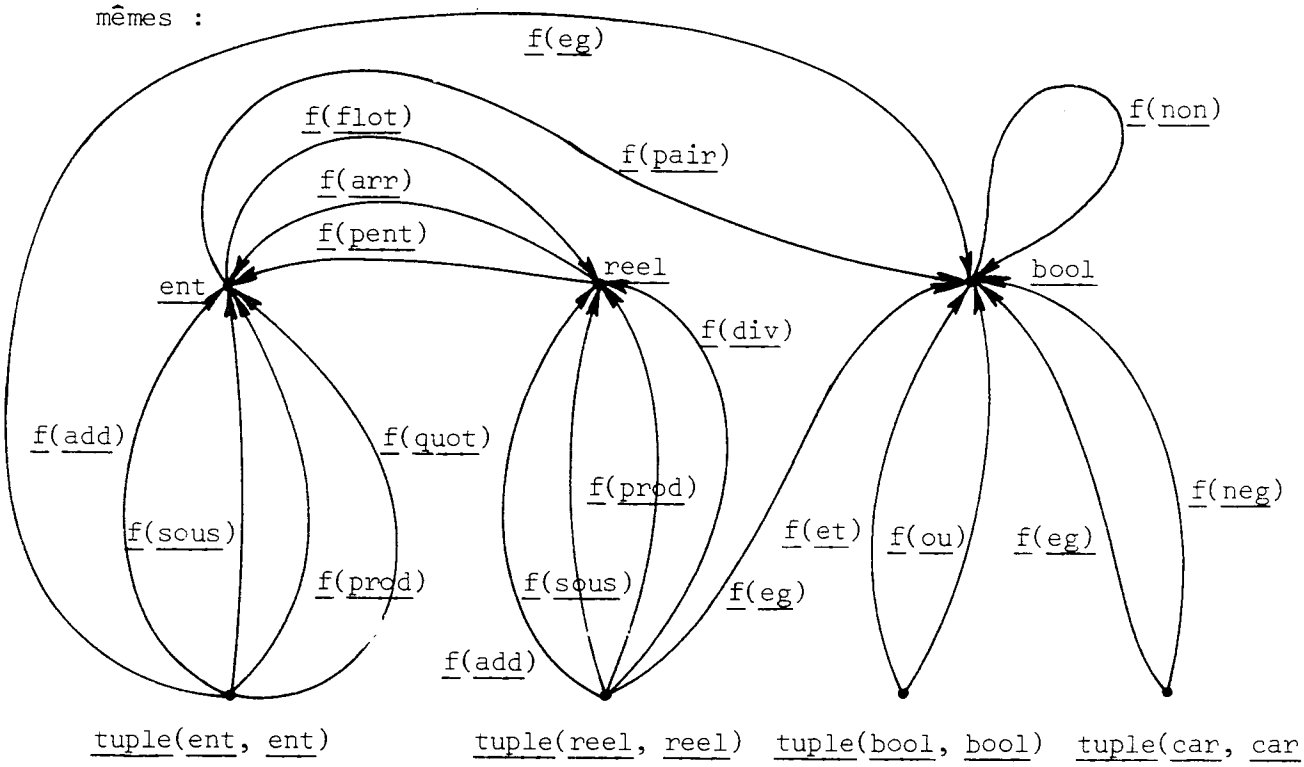
une fonction g (opération ou procédure) qui prend un paramètre de type t1 et qui retourne un résultat de type t2, le graphe des propriétés pour les modes de base en Algol 68 contient ces deux sortes d'arcs. Par exemple, un sous-graphe de ce graphe est :



D'autre part, en Algol 68, un certain nombre d'autres modes, comme compl, sont intégrés aux modes de base, bien que leur définition soit faite, de façon standard, au moyen d'une déclaration de mode. Un tel mode peut alors être mis dans le graphe des propriétés, avec les arcs correspondants. Dans le cas de compl, l'arc le plus "significatif" de cette intégration aux modes de base est :



De même, en BASEL, un sous-graphe du graphe des propriétés des modes de base contient un certain nombre d'autres modes que les modes de base eux-mêmes :



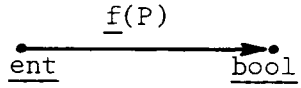
On remarque qu'en BASEL, il n'y a dans ce graphe que des arcs représentant des opérations : il n'y a en BASEL aucune conversion automatique entre modes de base.

La seule possibilité qui soit offerte de modifier ce graphe en BASEL est de définir des procédures, simples ou génériques. En Algol 68, c'est de définir des procédures ou des opérateurs - qui ne sont en fait qu'un "sucre syntaxique" pour des procédures génériques restreintes -. Ainsi,

en Algol 68, une déclaration de procédure comme :

proc P = (ent A) bool : A > 10

introduit le nouvel arc suivant dans le graphe :



Si m est un mode construit, dans un système de modes classiques, toutes les propriétés caractéristiques des objets de mode m dépendent exclusivement :

- du constructeur utilisé ;
- des modes et des objets divers pris comme arguments.

Exemple, en BASEL :

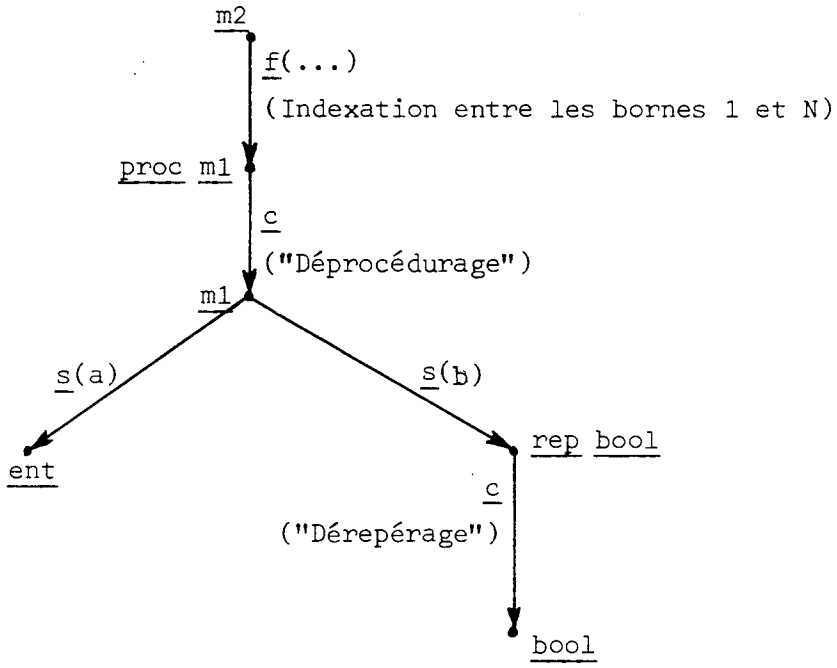
Soit le mode m = struct (ent A, var bool B)

- Le constructeur utilisé, struct, spécifie la configuration statique des objets de mode m : se sont des structures, avec les mécanismes de construction que cela implique. Il spécifie aussi le comportement dynamique caractéristique des structures : on peut les utiliser dans une sélection. Tout cela est imposé, implicitement, par l'utilisation de struct.
- Les arguments ent, A, bool, et B spécifient que le champ A des structures de mode m ont les propriétés caractéristiques des entiers, et que le champ B a les propriétés caractéristiques des objets de mode var bool.
- A son tour, le constructeur var impose la configuration statique et le comportement dynamique des variables : on peut leur affecter des valeurs et elles peuvent subir l'extraction de valeur. Quant au mode bool, il indique les propriétés caractéristiques des valeurs que peuvent retenir ces variables.

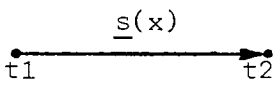
On voit donc qu'un tel mécanisme de définition de mode par des constructeurs procède de façon strictement hiérarchique : on construit un mode à partir d'autres modes qui sont eux-mêmes construits à partir d'autres modes, etc... jusqu'aux modes de base. D'ailleurs, le graphe des propriétés pour les objets d'un mode construit à la forme d'un arbre. En effet, si l'on reprend un exemple donné plus haut en Algol 68 :

mode m1 = struct (ent a, rep bool b)
mode m2 = [1 : N] proc m1

Le graphe est :



où un arc de la forme :



est une information sur la configuration statique des objets de type t1 : il indique qu'un objet de type t1 est construit avec un certain nombre d'éléments, l'un d'entre-eux, de type t2, pouvant être obtenu par une opération de sélection utilisant le sélecteur x.

Donc, un mode construit a toutes ses propriétés caractéristiques décrites dans un arbre, et la seule possibilité de sortir de la structure hiérarchique est offerte par les définitions de procédures.

Si l'on considère maintenant les ambitions des langages extensibles dans le domaine de la sémantique, il ne semble pas que ce genre de mécanisme hiérarchique de définition de modes soit suffisant. En effet, si un langage extensible ne doit en aucun cas préjuger ni de la nature des objets utilisés dans un langage étendu, ni des opérations sur ces objets, le choix imposé d'un certain nombre de modes de base ayant des propriétés prédéfinies ne va certainement pas dans le sens voulu. Ces modes de base sont en effet introduits dans le langage exactement de la même façon que

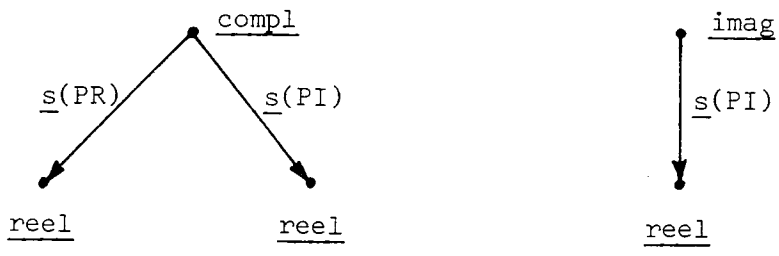
cela serait fait pour un langage non extensible. Ceci se traduit d'ailleurs très bien dans le graphe des propriétés correspondant :

- on ne peut pas lui rajouter de nouveaux sommets, c'est-à-dire "inventer" de nouveaux modes de base pour des objets ayant des notations et des représentations internes que l'on choisirait.
- on ne peut pas lui rajouter d'arcs, sauf par le biais de définitions de procédures. Entre autres, on ne peut pas "inventer" de nouvelles conversions.

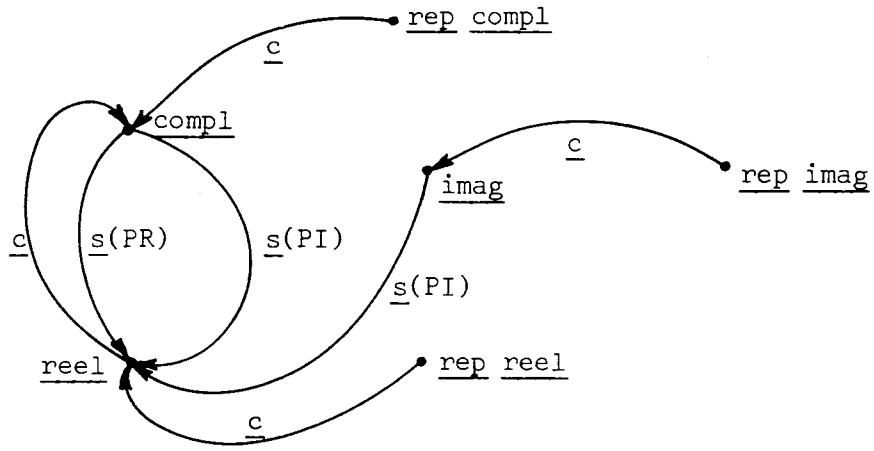
D'autre part, le mécanisme hiérarchique des constructeurs impose un répertoire figé d'opérations générales qui peuvent s'appliquer aux objets d'un mode construit, comme par exemple, en BASEL, l'extraction de valeur, l'appel implicite, la sélection, l'indexation, l'affectation, etc..., opérations qui sont traduites dans l'arbre qui constitue le graphe des propriétés pour un mode construit. Aucun moyen, autre que des procédures classiques, n'est prévu pour transformer cet arbre en un graphe plus général, en "inventant", par exemple, de nouveaux constructeurs ou de nouvelles conversions. Un exemple, pris en Algol 68, montre que parfois il serait souhaitable de pouvoir faire cela :

```
(mode compl = struct (reel PR, reel PI) ;
     compl Z ; reel X ;
          Z := X ; C ceci est autorisé C
     mode imag = struct (reel PI) ;
     imag I ;
          Z := I ; C ceci est interdit C ...)
```

Les arbres correspondant aux modes compl et imag sont :



En fait, si l'on rattache ces arbres au graphe des modes de base, un graphe plus complet est obtenu, où le sommet correspondant à réel devient unique :



Ce graphe est celui des propriétés des objets utilisés dans le programme particulier ci-dessus.

L'affectation Z := X est donc permise grâce au "chemin" de conversion qui va de rep réel à compl. Mais Z := I est interdit car, s'il y a bien un arc de conversion de rep imag à imag, aucun chemin de conversion ne va d'imag à compl. Le mode imag n'étant pas "standard", il est bien sûr normal que cette conversion n'existe pas, car tout ne peut pas être prévu dans la définition originelle du langage. Mais aucun mécanisme ne permet de la définir si on a besoin d'elle.

Donc, le mécanisme des modes classiques, utilisé en BASEL, en Algol 68 et, dans une certaine mesure, en PPL, accompagné seulement par un mécanisme de procédures, réalise une extensibilité sémantique embryonnaire. Cela peut être suffisant dans bien des cas. Mais, en restant toujours raisonnablement à la portée d'un utilisateur qui définit lui-même certaines des caractéristiques du langage dont il va se servir, on peut introduire quelques outils supplémentaires qui élargissent les possibilités de ce mécanisme de modes : c'est le but du mécanisme des propriétés additionnelles.

4.3. LES PROPRIETES ADDITIONNELLES

Avec le mécanisme des modes classiques, tout ce qui concerne la configuration statique et un certain nombre d'aspects du comportement dynamique sont décrits, en général, de façon implicite par les modes de base et les constructeurs.

Le mécanisme proposé maintenant permet, en partant de modes définis de façon classique, de rajouter aux objets d'un mode donné un certain nombre de propriétés relatives à leur comportement dynamique. La définition de ces propriétés additionnelles est faite de façon explicite au moment de la définition du mode.

Le principe de base de ce mécanisme a été établi par moi-même, et je l'ai décrit sous une première forme dans [Sc70]. De plus, j'ai proposé dans [Jo73] l'introduction en Algol 68 d'un système fondé sur le même principe.

Ce mécanisme se présente de façon externe sous la forme d'une déclaration, dite "déclaration de mode complète".

4.3.1. LA DECLARATION DE MODE COMPLETE -

On pourrait envisager la déclaration de mode complète comme un mécanisme d'extension sémantique qui utiliserait BASEL comme langage de base. Cependant, afin de faire ressortir l'essentiel du mécanisme présenté ici et d'en simplifier l'exposé, on utilisera comme langage de base le langage BASELO qui est obtenu en faisant subir à BASEL quelques transformations importantes qui concernent surtout le traitement des modes.

Il est sans intérêt de donner ici une définition complète et rigoureuse de BASELO. Sa syntaxe est, pour l'essentiel, la même que celle de BASEL, et ses particularités intéressantes pour ce qui va suivre sont :

- Il y a quatre modes de base : ent, bool, reel, car.
- Il y a quatre constructeurs : var, proc, tuple, seq.
- La relation d'égalité générale entre deux modes n'est pas définie dans la définition du langage.
- La relation de compatibilité entre deux modes n'est pas définie dans la définition du langage.
- Le principe d'un mécanisme de modifications automatiques existe, mais aucune modification élémentaire n'est définie dans la définition du langage.
- Trois nouvelles opérations sont introduites :

  ex O, qui a pour résultat la valeur de mode m extraite du résultat de mode var m de l'opérande O.

  ap O, qui a pour résultat le résultat de mode m, s'il existe, de l'appel de la procédure de mode proc m, résultat de O.

  dt O, qui a pour résultat la valeur de mode m, unique élément du tuple de mode tuple (m), résultat de O.

- La relation d'égalité, la relation de compatibilité et les modifications élémentaires sont établies par la déclaration de mode, qui est ici la déclaration de mode complète.

Les autres caractéristiques de BASELO sont soit des conséquences de ces quelques points, soit sans intérêt particulier pour la suite. (Cependant, pour simplifier, on ne considérera pas l'éventualité de déclarations de modes récursifs).

4.3.1.1. Syntaxe de la déclaration de mode complète

La syntaxe de la déclaration de mode complète, selon les conventions définies en 3.1.1.1 pour la syntaxe de BASEL, est la suivante :

declaration-de-mode complete : symbole rep mode [proprietes]
proprietes : predicat|
transformations
predicat : si fonction
transformations : avec <conversion|selecteur> <conversion|selecteur> ...
conversion : <de|vers> mode par fonction
selecteur : sel mode identificateur par fonction
fonction : < identificateur dans partie-instruction >

Dans cette syntaxe, les non-terminaux :

partie-instruction
mode
identificateur
symbole

sont ceux définis au chapitre 3, dans la description de BASEL.

4.3.1.2. Sémantique de la déclaration de mode complète

Soient :

- S un symbole,
- M, M1, M2, ..., Mn, N1, N2, ..., Np, P1, P2, ..., Pq des modes,
- F, F1, F2, ..., Fn, G1, G2, ..., Gp, H1, H2, ..., Hq des fonctions,
- I, I1, I2, ..., In, J1, J2, ..., Jp, K1, K2, ..., Kq, L1, L2, ..., Lq des identificateurs,
- A, A1, A2, ..., An, B1, B2, ..., Bp, C1, C2, ..., Cq des parties instructions.

Plusieurs cas sont à considérer pour décrire l'effet de la déclaration de mode complète.

1er cas

Si une déclaration de mode complète a la forme :

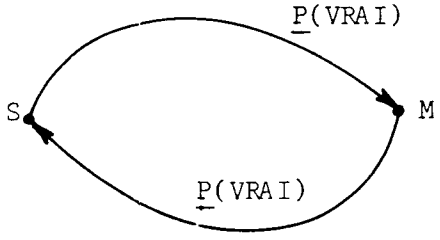
S rep M

cette déclaration a en fait exactement la même signification que la déclaration de mode en BASEL : les objets de mode S ont la même configuration statique et le même comportement dynamique que les objets de mode M.

Dans le graphe des propriétés, cette déclaration crée un nouveau sommet, représentant le mode S, et deux arcs :

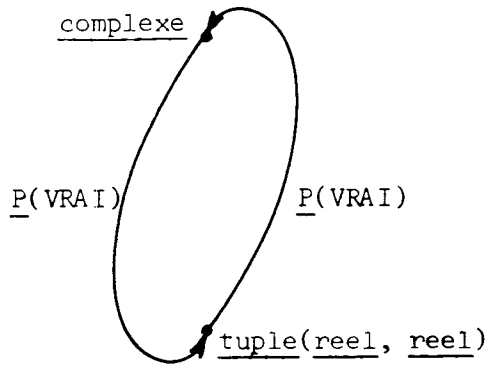
- un arc, qui va du sommet représentant S au sommet représentant M, et qui représente un prédicat identiquement vrai, avec l'interprétation : "il est toujours vrai qu'un objet de mode S peut être utilisé dans un contexte qui exige un objet de mode M" ;
- un arc, qui va en sens inverse du précédent, et qui représente lui aussi un prédicat identiquement vrai, avec l'interprétation : "il est toujours vrai qu'un objet de mode M peut être utilisé dans un contexte qui exige un objet de mode S".

Ceci est représenté graphiquement par :



Exemple :

complexe rep tuple (reel, reel)



2ème cas

Si une déclaration de mode complète a la forme :

S rep M si F

où F est la fonction :

<I dans A>

il faut que A ait un résultat de mode bool. F est alors transformé en un texte de procédure F' qui a la forme :

<I est M dans A>

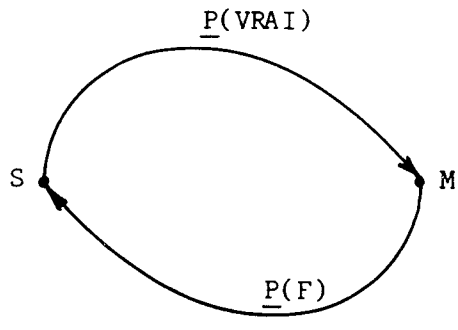
La signification de cette déclaration est alors la suivante :

- les objets de mode S acquièrent la configuration statique et le comportement dynamique des objets de mode M.
- Un objet X de mode M ne peut être considéré comme un objet de mode S que si F'(X) est vrai.

Dans le graphe, cette déclaration crée un nouveau sommet, qui représente le mode S, et deux arcs :

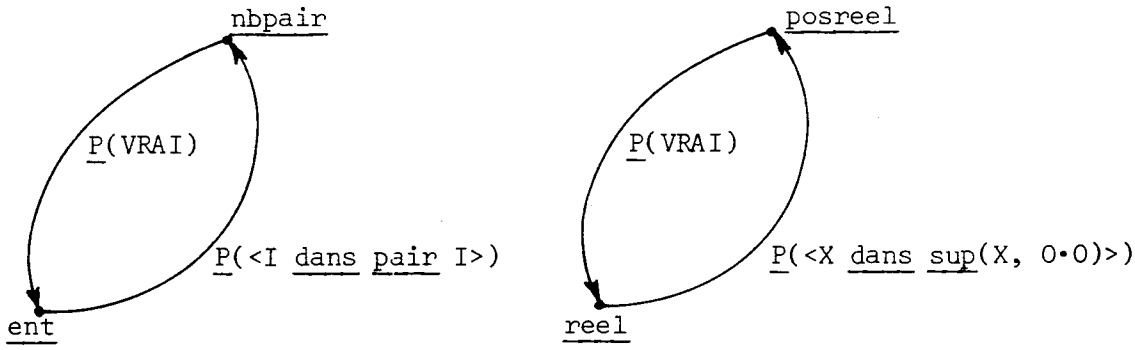
- un arc, qui va du sommet représentant S au sommet représentant M, et qui représente un prédicat identiquement vrai, avec l'interprétation : "il est toujours vrai qu'un objet de mode S peut être utilisé dans un contexte qui exige un objet de mode M".
- un arc, qui va en sens inverse du précédent, et qui représente un prédicat avec l'interprétation suivante : "si un objet X de mode M est utilisé dans un contexte qui exige un objet de mode S, alors il faut que F'(X) soit vrai".

Ceci est représenté graphiquement par :



Exemples :

nbpair rep ent si <I dans pair I>
posreel rep reel si <X dans sup (X, 0.0)>



3ème cas

Si une déclaration de mode complète a la forme :

S rep M avec de M1 par F1
de M2 par F2
...
de Mn par Fn

où Fi ($1 \leq i \leq n$) est la fonction :

<Ii dans Ai>

il faut que Ai ait un résultat de mode M. Fi est alors transformé en un texte de procédure F'i qui a la forme :

<Ii est Mi dans Ai>

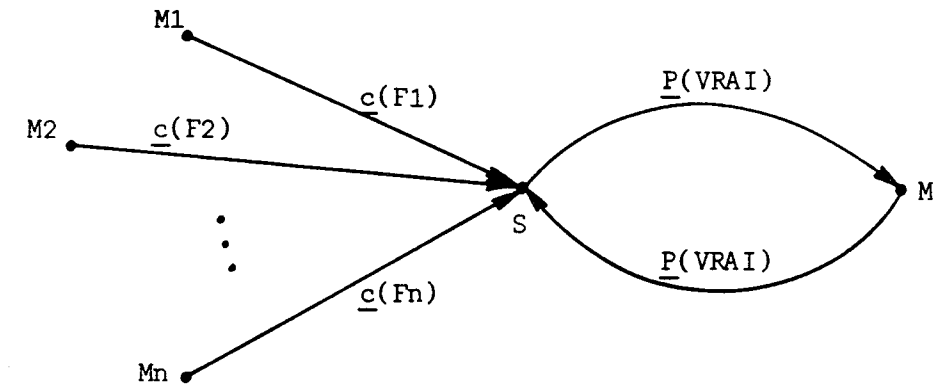
La signification de cette déclaration est la suivante :

- les objets de mode S acquièrent la configuration statique et le comportement dynamique des objets de mode M.
- Un objet X de mode Mi ($1 \leq i \leq n$) peut être converti en un objet de mode S, la conversion effectuant les actions définies par F'i (X).

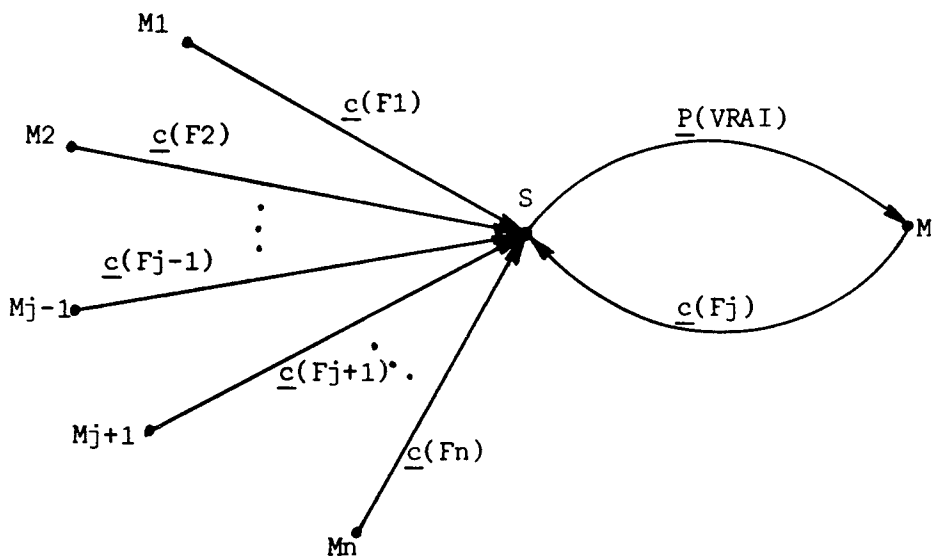
Dans le graphe, cette déclaration crée un sommet, qui représente le mode S et, selon le cas, elle crée soit n+1, soit n+2 arcs :

- un arc, qui va du sommet représentant S au sommet représentant M, et qui représente un prédicat identiquement vrai, avec l'interprétation : "il est toujours vrai qu'un objet de mode S peut être utilisé dans un contexte qui exige un objet de mode M".
- si M $\notin$ {M1, M2, ..., Mn}, un arc, qui va du sommet représentant M au sommet représentant S, et qui représente un prédicat identiquement vrai.
- n arcs, le $i^{ème}$ d'entre eux allant du sommet qui représente Mi au sommet qui représente S, et qui représente une conversion, avec la signification : "Si un objet X de mode Mi est utilisé dans un contexte qui exige un objet de mode S, alors cet objet est remplacé par le résultat de F'i (X)".

Ceci est représenté graphiquement soit par :



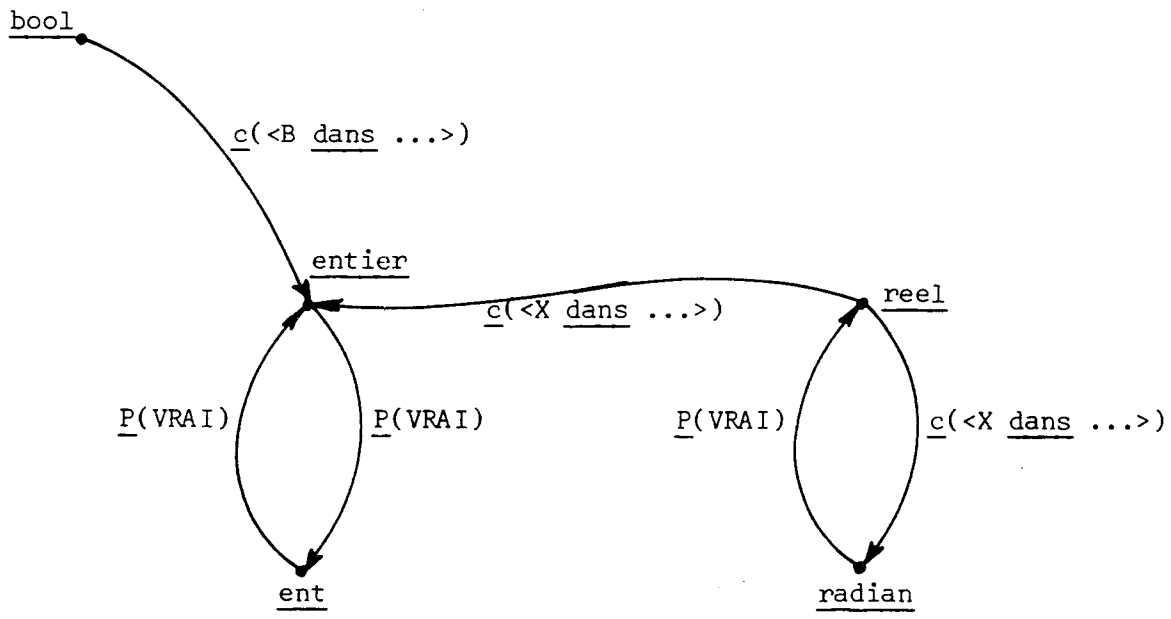
soit par :



Exemples :

```
entier rep ent
     avec de reel par <X dans pent (X)>
          de bool par <B dans B → 1|0>,

radian rep reel
     avec de reel par <X dans
                       (Y est reel, PI est reel, DPI est reel dans
                       PI = 3.14159 ; DPI = prod (PI, 2.0) ; Y = X ;
                       L : sup (abs Y, PI)
                       → Y = sous (Y, prod(DPI, flott sign Y)) ;
                       allera L|Y)>
```



4ème cas

Si une déclaration de mode complète a la forme :

S rep M avec vers N1 par G1
vers N2 par G2
.
.
.
vers Np par Gp

où Gi ($1 \leq i \leq p$) est de la forme :

<Ji dans Bi>

il faut que Bi ait un résultat de mode Ni. Gi est alors transformé en un texte de procédure G'i qui a la forme :

<Ji est M dans Bi>

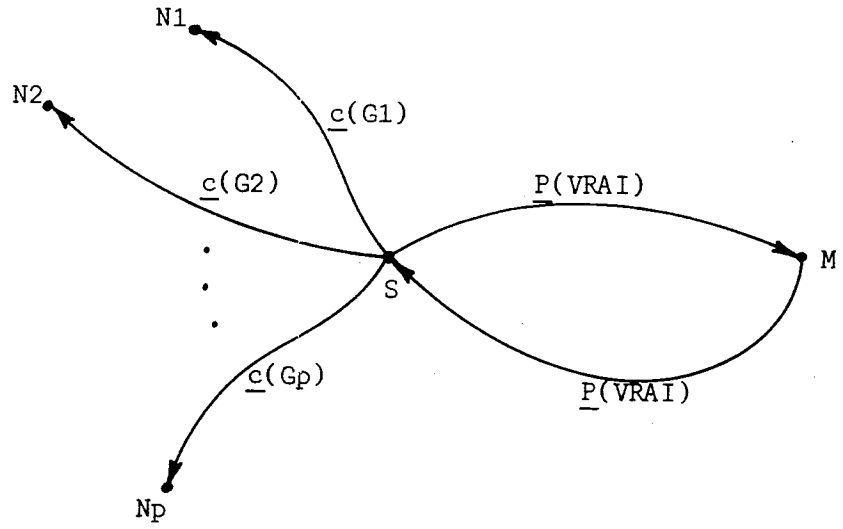
La signification de cette déclaration est la suivante :

- Les objets de mode S acquièrent la même configuration statique et le même comportement dynamique que les objets de mode M.
- Un objet X de mode S peut être converti en un objet de mode Ni ($1 \leq i \leq p$), la conversion effectuant les actions définies par G'i (X).

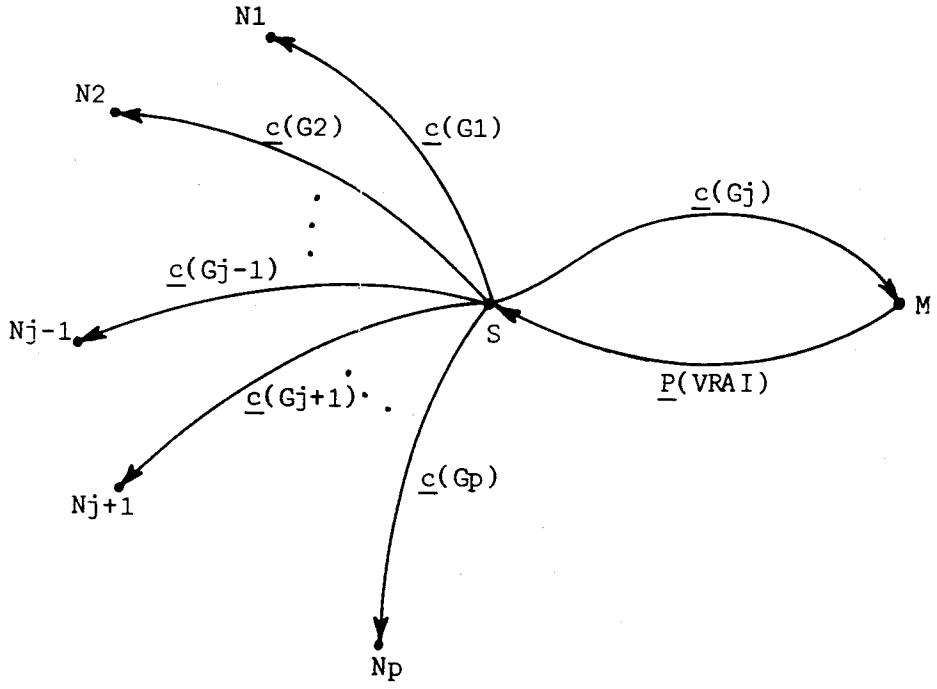
Dans le graphe, cette déclaration crée un sommet, qui représente le mode S, et, selon les cas, elle crée p+1 ou p+2 arcs :

- si $M \notin \{N1, N2, \ldots, Np\}$, un arc qui va du sommet représentant S au sommet représentant M, et qui représente un prédicat identiquement vrai.
- un arc, qui va en sens inverse du précédent, et qui représente un prédicat identiquement vrai.
- p arcs, le $i^{ème}$ d'entre eux allant du sommet qui représente S au sommet qui représente Ni, et qui représente une conversion, avec la signification suivante : "Si un objet X de mode S est utilisé dans un contexte qui exige un objet de mode Ni, alors cet objet est remplacé par le résultat de G'i (X)".

Ceci est représenté graphiquement, soit par :



soit par :



5ème cas

Si une déclaration de mode complète est de la forme :

S rep M avec sel P1 L1 par H1
sel P2 L2 par H2
...
sel Pq Lq par Hq

où Hi ($1 \leq i \leq q$) est de la forme :

<Ki dans Ci>

il faut que Ci ait un résultat de mode Pi. Hi est alors transformé en un texte de procédure H'i qui a la forme :

<Ki est M dans Ci>

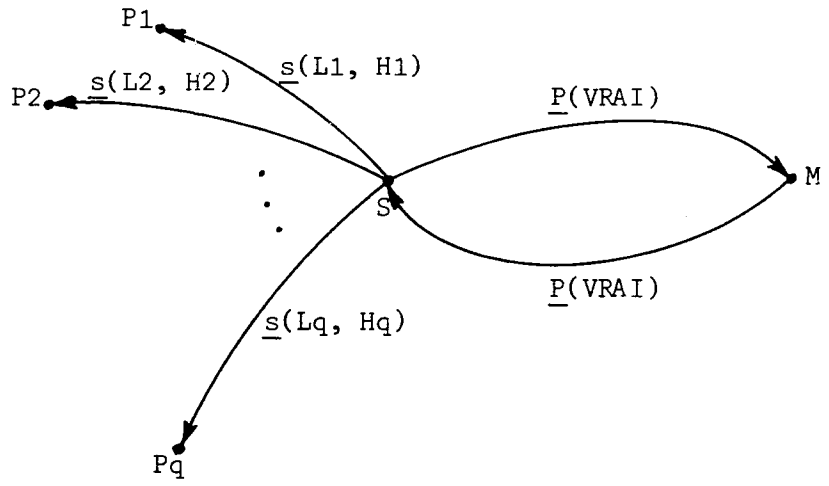
La signification de cette déclaration est la suivante :

- Les objets de mode S ont la même configuration statique et les mêmes propriétés dynamiques que les objets de mode M.
- Un objet X de mode S peut être utilisé dans une sélection, de la forme Li de S ($1 \leq i \leq q$), l'effet de cette opération étant défini par H'i (X).

Dans le graphe, cette déclaration crée un sommet, qui représente le mode S, et q+2 arcs :

- Un arc, qui va du sommet représentant S au sommet représentant M, et qui représente un prédicat identiquement vrai.
- Un arc, qui va en sens inverse du précédent, et qui représente un prédicat identiquement vrai.
- q arcs, le $i^{ème}$ d'entre eux allant du sommet qui représente S au sommet qui représente Pi, et qui représente un sélecteur, avec la signification suivante : "Un objet de mode Pi peut être obtenu à partir d'un objet de mode S en appliquant à cet objet x de mode S la sélection Li de X, sélection qui a pour effet H'i (X)".

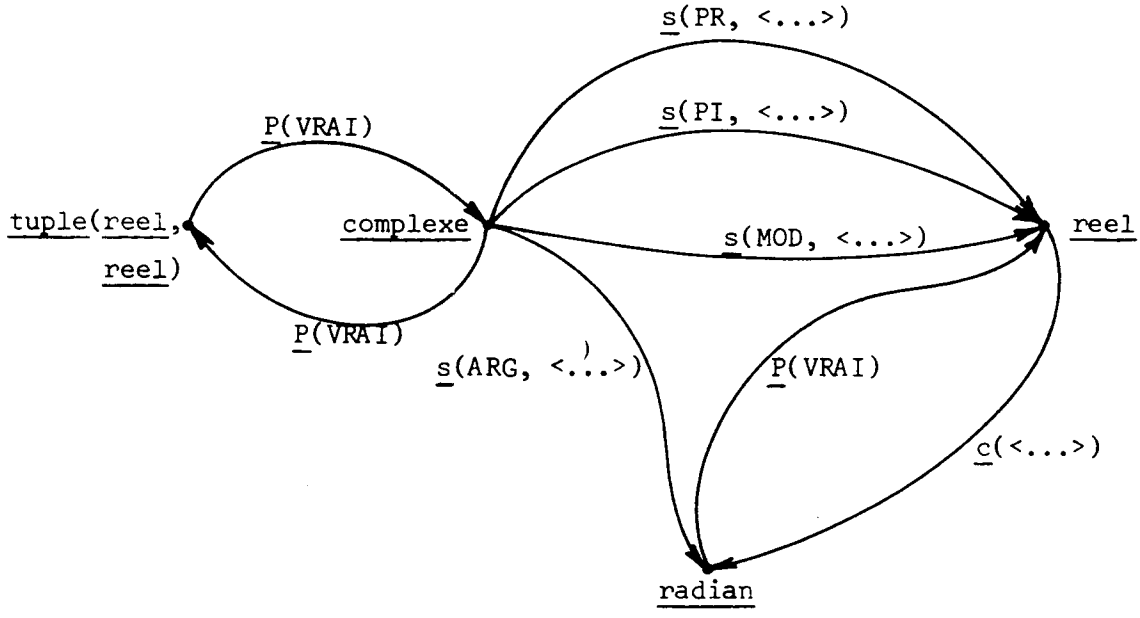
Ceci est représenté graphiquement par :



Exemple :

```
complexe rep tuple (reel, reel) avec
    sel reel PR par <Z dans <A est reel, B est reel dans A> Z>
    sel reel PI par <Z dans <A est reel, B est reel dans B> Z>
    sel reel MOD par <Z dans <A est reel, B est reel dans
                              RAC2 (add (puiss (A, 2),
                                         puiss (B, 2)))> Z>
    sel radian ARG par <Z dans <A est reel, B est reel dans
                              radian ARTAN (div (B, A))> Z>
```

Le sous-graphe concernant cette déclaration est :



Autres cas

Etant donné la syntaxe de la déclaration de mode complète, les propriétés pouvant être définies sont :

- soit un prédicat,
- soit un mélange quelconque de conversions et de sélecteurs.

Dans le cas où il s'agit d'un mélange de conversions, du mode en cours de définition et vers ce mode, et de sélecteurs, la signification de la déclaration est une combinaison triviale de ce qui vient d'être dit, avec les effets correspondants sur le graphe des propriétés.

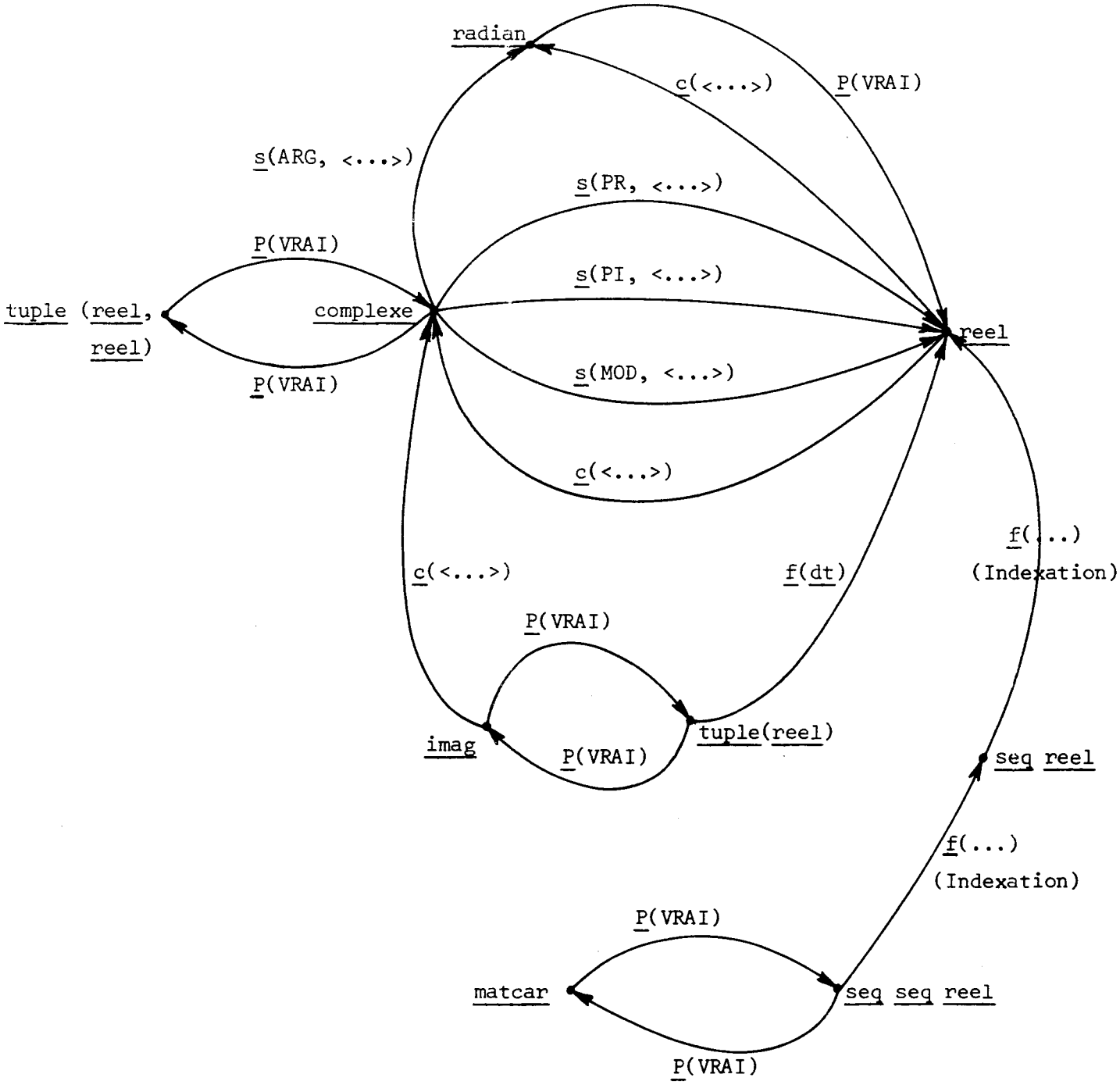
Exemples :

```
complexe rep tuple (reel, reel) avec
     sel reel PR par <...>
     sel reel PI par <...>
     sel reel MOD par <...>
     sel radian ARG par <...>
     de reel par <X dans [X, 0.0]>,

imag rep tuple (reel) avec
     sel reel PI par <I dans dt I>
     vers complexe par <I dans [0.0, dt I]>,

matcar rep seq seq reel avec
     si <M dans (I est ent, J est ent dans
                 I = 0 ; J = long M ;
                 L : eg (I, J) → vrai|
                 I = add (I, 1) ; eg (long M I, J) → allera L|faux)>
```

Le sous-graphe concernant ces déclarations est :



4.3.2. ROLE DU GRAPHE DANS LA SEMANTIQUE STATIQUE

Les règles du traitement des modes dans la sémantique statique sont toutes définies par le graphe des propriétés.

Si l'on admet qu'il y a en BASELO un certain nombre de contexte - les mêmes qu'en BASEL - qui "exigent" une valeur d'un mode donné, la sémantique statique de BASELO définit seulement que des modifications peuvent être déclenchées dans ces contextes, mais elle ne dit pas quelles sont ces modifications : c'est la déclaration de mode complète qui définit des modifications de base, et l'information correspondante et contenue dans le graphe des propriétés.

En effet, ce graphe définit :

- une relation de compatibilité entre modes,
- une relation d'égalité entre modes,
- un mécanisme particulier de modifications, comprenant une description des actions qu'effectue chaque modification de base.

4.3.2.1. Définitions préliminaires

Pour décrire l'utilisation du graphe par la sémantique statique, il convient d'abord d'introduire quelques conventions qui servent à cette description.

- Le changement de mode -

Soient deux modes m et m' représentés par deux sommets M et M', tels qu'il existe de M à M' un arc représentant un prédicat. Etant donné la façon dont sont définis les prédicats, il ne peut y avoir plus d'un tel arc allant de M à M'.

Le processus statique de détermination des modifications à appliquer dans un contexte donné peut alors, dans une instruction transformée qu'il construit, utiliser une nouvelle possibilité syntaxique des "bases", le changement de mode, possibilité inaccessible à un utilisateur :

base : accès |
texte-de-procédure |
notation |
expression-composée |
changement de mode

avec :

changement-de-mode : / mode * accès /

Le changement de mode a un rôle uniquement dans la sémantique statique et ne provoque aucune action dans la sémantique dynamique. Ce rôle est le suivant :

- l'accès produit une valeur de mode m, représenté par le sommet M du graphe ;
- le mode spécifié dans le changement de mode est m' , représenté par le sommet M' du graphe ;
- la valeur produite par l'accès est alors, sans aucune action particulière, simplement considérée comme étant maintenant une valeur de mode m'.

- L'erreur -

Dans une instruction transformée construite par le processus des modifications, la valeur nulle peut prendre une forme dont l'utilisation est réservée à ce processus : c'est l'erreur.

valeur-nulle : nul mode |
erreur

avec :

erreur : erreur mode

Si m est le mode spécifié, le résultat de l'erreur est de mode m. Cependant, les actions effectuées dynamiquement ne sont pas définies ici : ce peut être, par exemple, un arrêt "brutal" de l'exécution du programme, ou encore l'impression d'un message suivie de l'obtention, en guise de résultat, de la valeur nulle de mode m.

4.3.2.2. Fonctions de passage

Soient deux modes, m et m', représentés dans le graphe par les sommets M et M'.

Si ces deux sommets ne sont pas confondus, et s'il existe au moins un chemin de M à M' ne contenant que des arcs qui représentent des prédicats et des conversions, soit C l'un de ces chemins, que l'on choisit élémentaire.

On considère d'abord le cas où C est de longueur 1 :

- Si l'unique arc de C représente un prédicat, on appelle fonction de passage f de m à m' une procédure construite de la façon suivante :

  . Si le prédicat est identiquement vrai, cette procédure est :

  f = <X est m dans/m'*X/>

  . Si le prédicat, n'est pas identiquement vrai, cette procédure est :

  f = <X est m dans F'(X)→/m'*X/|erreur m'>

  où F' est la procédure de la forme :

  <I est m dans A>

  obtenue à partir de la fonction F de la forme <I dans A> qui définit le prédicat.

- Si l'unique arc de C représente une conversion, on appelle fonction de passage f de m à m' la procédure suivante

  f = <X est m dans D>

  qui est directement obtenue à partir de la fonction <X dans D> qui définit la conversion.

On considère ensuite les cas où C est de longueur $\ell > 1$. Soient alors :

m1, m2, ..., mℓ+1, avec m1 = m et mℓ+1 = m'

la suite des modes correspondant aux sommets par lesquels passe ce chemin. Chacun des arcs qui composent ce chemin définit une fonction de passage : l'arc ai, qui va de mi à mi+1 ($1 \leq i \leq \ell$) définit la fonction de passage fi. La fonction de passage de m à m' est alors la composition fonctionnelle :

$$f = \text{<X est m dans } f_\ell\ (f_{\ell-1}\ (\ldots\ f_2\ (f_1\ (X))\ \ldots))\text{>}$$

4.3.2.3. Relations entre les modes

Soient deux modes m et m'.

- S'il n'existe pas de fonction de passage ni de m à m', ni de m' à m, il n'y a aucune relation entre m et m'.
- S'il existe une et une seule fonction de passage de m à m', alors on dit que m est compatible avec m', ce qui se note :

$$m \Rightarrow m'$$

- S'il existe une fonction de passage de m à m', et que cette fonction de passage soit entièrement construite à base de prédicats identiquement vrais, on dit que m est statiquement compatible avec m', ce qui est noté :

$$m \Rightarrow\!\!\!> m'$$

  Etant donné la façon dont sont établis les prédicats, cette fonction de passage est nécessairement unique et il est clair que la relation $m \Rightarrow\!\!\!> m'$ implique la relation $m \Rightarrow m'$.
- Si on a à la fois la relation $m \Rightarrow\!\!\!> m'$ et la relation $m' \Rightarrow\!\!\!> m$, on dit que m est égal à m' :

$$m = m'$$

- S'il existe plusieurs fonctions de passage de m à m' et si au moins l'une d'entre elles n'est pas construite entièrement à base de prédicats identiquement vrais, alors il y a une ambiguité (voir le paragraphe 4.4.1.7) dans le système de modes défini par les déclarations de modes complètes qui ont été faites.

4.3.2.4. Mise en oeuvre des modifications

Les relations entre les modes et les fonctions de passage sont utilisées par le mécanisme des modifications pour décider de la façon dont une valeur de mode m peut être employée dans un contexte qui exige une valeur de mode m'.

- S'il n'est pas vrai que $m \Rightarrow m'$, le programme est erroné.

- Si m $\Rightarrow$ m', l'expression E qui a un résultat de mode m est remplacée par l'expression :

$$f(E)$$

où f est la fonction de passage de m à m'.

- Si m $\Rightarrow\!\!>$ m', cette transformation de E ne rajoute pas d'actions à l'exécution de E, telle qu'elle est définie par la sémantique dynamique.

4.3.3. REMARQUES SUR CE MECANISME

Le mécanisme des propriétés additionnelles permet d'établir certaines des bases de la sémantique statique d'un langage de programmation.

Avec ce mécanisme, et en particulier grâce aux conversions et aux prédicats, on peut donner au graphe des propriétés pour les modes construits une structure plus générale que l'arbre auquel on était limité par le mécanisme des modes classiques.

Cependant, le graphe des propriétés pour les modes de base n'est toujours pas susceptible d'être ni construit ni modifié, et on ne peut espérer, par ce mécanisme, construire un mécanisme de modification qui aurait la généralité de celui de BASEL ou d'Algol 68, au moins dans son principe, car le nombre de modes impliqués dans le graphe est nécessairement fini et chaque propriété individuelle doit toujours être explicitement construite : on ne peut donc pas, par exemple, définir l'"extraction de valeur" de BASEL dans toute sa généralité, car elle concerne, potentiellement, une infinité de modes.

Le mécanisme des classes, qui part sur des notions plus primitives, et qui ne suppose ni la préexistence de modes de base ni le support d'un système hiérarchique de construction de modes, permet d'aller plus loin dans cette recherche d'une plus grande généralité.

4.4. LE MECANISME DES CLASSES

Décrit à l'origine dans [Jo71], puis présenté sous une forme primitive mais avec plus de détails dans [Jo72] et dans la thèse de D. Bert [Be73], le mécanisme des classes s'inscrit dans la ligne générale des travaux les plus récents sur le problème des types et de la définition de la sémantique des langages de programmation. En effet, Wegbreit dans sa thèse [We7o] et dans [We74], Alice et Mike Fisher dans [Fi73], Barbara Liskov dans [Li74], Morris dans [Mo74], Reynolds dans [Re69] et dans [Re74] proposent tous des mécanismes de types qui s'appuient sur des bases formelles analogues et qui rejoignent eux aussi les problèmes plus généraux de la sémantique.

Je décris d'abord ici le mécanisme primitif des classes, avec quelques améliorations par rapport à la première forme que je lui avais donnée, pour introduire les idées de base, puis j'indique ensuite les diverses façons dont j'envisage de généraliser ce mécanisme vers une véritable extensibilité de la sémantique d'un langage. La direction générale des travaux dont ce mécanisme fait actuellement l'objet est présentée dans une publication récente [Sc74] et un document de travail sur ce sujet [Jo74] a été soumis au WG2.1 (groupe de travail de l'IFIP sur Algol) en guise de proposition pour un point de départ éventuel des activités futures de ce groupe.

4.4.1. LE MECANISME PRIMITIF

L'idée de départ du mécanisme des classes est très simple et fait directement usage de l'essence même de la notion de mode dans un langage comme BASEL ou Algol 68 : un mode est une classe d'objets qui ont un certain nombre de propriétés en commun.

4.4.1.1. Principes de départ

Le mécanisme primitif des classes, introduit dans [Jo71], [Jo72] et [Be73] permet donc :

- La déclaration de classes, considérées comme des ensembles, ou "hôtes" pour éléments potentiels, avec leurs relations éventuelles avec d'autres classes, exprimées en termes d'inclusion et de produit cartésien.

- La définition explicite de propriétés pour les éléments (objets, valeurs) d'une classe, considérées comme des applications d'une classe dans une autre, et exprimées par des fonctions dans un langage de programmation. Ces applications peuvent être de trois sortes :

    . des fonctions, dont l'appel doit toujour être explicitement écrit pour avoir lieu.

    . des conversions, dont l'appel est en général implicite, et provoqué par l'application des règles de la sémantique statique.

    . des prédicats, dont l'appel obéit aux mêmes règles que celui des conversions.

4.4.1.2. Le langage de base

De la même façon que dans les publications où il a été introduit, le mécanisme primitif est présenté ici comme étant un mécanisme d'extension d'un langage extensible dont le langage de base est le langage L0, sans types, décrit au paragraphe 3.2.

Ce langage remplit ici très clairement son rôle de langage de base, en servant d'interface entre des langages étendus où sont définies des classes, et la machine au niveau de laquelle la notion de type est inconnue.

On rappelle que dans ce langage, la syntaxe est entièrement sous forme fonctionnelle, et que la forme des déclarations est :

def (I, E)

où I est un identificateur que cette déclaration rend synonyme du résultat de l'expression E. C'est cette forme de déclaration qui est reprise par le mécanisme primitif des classes.

On rappelle aussi qu'un programme dans ce langage est structuré en modules, un module pouvant être explicitement écrit, ou construit dynamiquement par liaison des paramètres d'un lambda.

Un module a la forme :

module (D1, D2, ..., Dn, E)

où les Di sont des déclarations et E une expression.

Un lambda a la forme :

lambda (I1, I2, ..., In, E)

où les Ik sont des identificateurs qui constituent les paramètres formels, et E est une expression.

La construction d'un module par liaison des paramètres d'un lambda a la forme :

lier (L, E1, E2, ..., En)

où L est le lambda et les Ei des expressions qui constituent les paramètres effectifs.

L'évaluation d'un module est commandée par :

eval (M)

où M est soit un module directement écrit dans cette expression, auquel cas il s'agit de l'analogue d'un bloc, soit un identificateur déclaré au préalable comme synonyme d'un module, auquel cas il s'agit de l'analogue d'un appel de procédure sans paramètre, soit un module obtenu par liaison des paramètres d'un lambda, auquel cas il s'agit de l'analogue d'un appel de procédure avec paramètres.

On rappelle enfin que les opérations disponibles dans LO donnent un accès direct aux représentations des objets et à leur organisation dans un modèle de la mémoire : opérations sur les représentations internes, transformations de notations en représentations internes, rangements en mémoire, etc.... Ces opérations ignorent la notion de "type".

4.4.1.3. La définition des classes

Au départ, le langage de base étant "sans type", il n'existe aucune classe : le graphe des propriétés, dont on va encore faire usage ici, est donc vide. Ensuite, chaque définition de classe introduit un nouveau sommet dans le graphe et ce sommet peut être relié par des arcs de diverses natures à d'autres sommets du graphe, selon la façon dont la classe correspondante a été définie.

- Création d'une classe indépendante -

Une déclaration de la forme :

def (C, classe)

définit une nouvelle classe C, sans spécifier en même temps aucune propriété pour les éléments potentiels de cette classe. Dans le graphe, il y a simplement création d'un nouveau sommet :

•C

Exemple :

def (ENT, classe)

Ceci crée le sommet :

•ENT

- Création d'une classe avec relations d'inclusion -

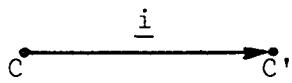
Il y a plusieurs façons de dire qu'une classe est incluse dans une autre au sens de l'inclusion d'un ensemble dans un autre.

Il y a d'abord l'inclusion simple, que l'on peut exprimer par :

def (C, in (C'))

Ceci signifie que tous les éléments de la nouvelle classe C sont aussi éléments de C', qu'ils ont donc toutes les propriétés qu'ont les éléments de C', mais qu'il sera possible de définir d'autres propriétés qui leur seront particulières.

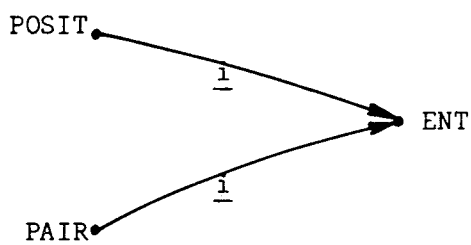
Dans le graphe, une telle déclaration crée un nouveau sommet, appelé C, et un arc qui va de C à C', qui représente l'inclusion de C dans C' et qui sera interprété comme un prédicat identiquement vrai : il est toujours vrai qu'un élément de C est aussi un élément de C' :



Exemple :

def (POSIT, in (ENT))
def (PAIR, in (ENT))

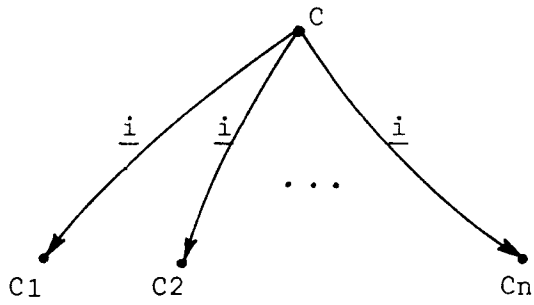
Le graphe devient alors :



Pour définir l'inclusion d'une nouvelle classe dans un ensemble d'autres classes, il y a l'intersection :

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

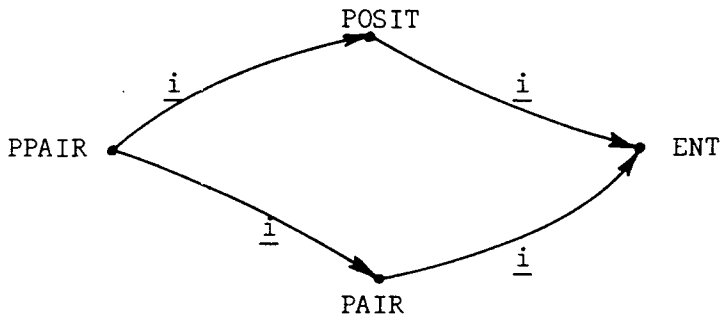
Dans le graphe, cette déclaration crée le nouveau sommet C, et des arcs indiquant que C est inclus dans C1, dans C2, ..., dans Cn :



Exemple :

def (PPAIR, inter (POSIT, PAIR))

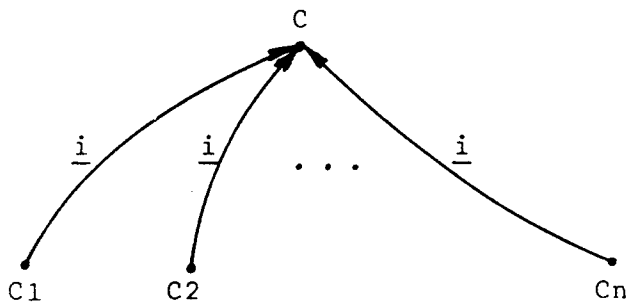
Le graphe est maintenant :



On peut aussi, grâce à l'union, définir une nouvelle classe qui inclut un certain nombre d'autres classes :

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

Ceci rée le sommet C dans le graphe et un arc d'inclusion de chaque classe Ci ($1 \leq i \leq n$) vers C :

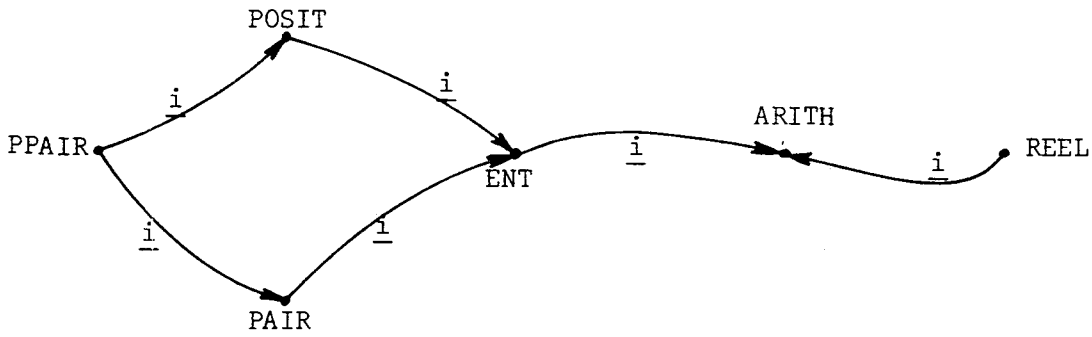


Exemple :

def (REEL, classe)

def (ARITH, union (ENT, REEL))

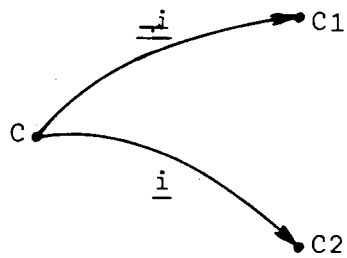
Nouvelle configuration du graphe :



Enfin, le complément permet de créer une classe C incluse dans une classe C2, et qui est le complément par rapport à C2 d'une autre classe C1 déjà incluse dans C2 :

def (C, compl (C1, C2))

Ceci crée le sommet C et deux arcs :

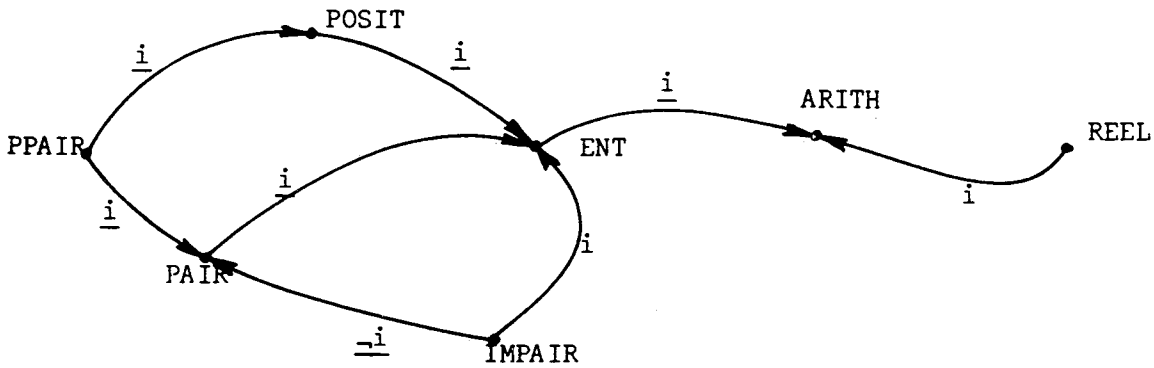


L'arc ¬i créé par cette déclaration sert seulement à détecter les classes qui seraient vides par construction.

En effet, si l'on déclare :

def (IMPAIR, compl (PAIR, ENT))

ceci donne au graphe la configuration :



Mais si l'on déclare ensuite :

def (XYZ, inter (PAIR, IMPAIR))

il peut être utile de savoir que cette classe ne peut contenir aucun élément. Pour détecter de telles classes on procède de la façon suivante [Be 72] :

- soit i (C, C') une relation qui est vraie si et seulement si un arc d'inclusion va de C à C'. Soit i^* la fermeture transitive de cette relation.
- soit K (C) l'ensemble des classes C' telles que i^* (C, C') est vrai :

$$K(C) = \{C' \mid i^*(C, C')\}$$

- soit n (C, C') une relation qui est vraie si et seulement si un arc avec la mention $\neg i$ va de C à C'.

Alors, une classe C est vide par construction si la proposition suivante est vraie :

$$\exists C1 \in K(C) \wedge \exists C2 \in K(C) \wedge n(C1, C2)$$

- Création d'une classe avec le produit cartésien -

Dans le mécanisme des classes, on distingue deux formes de produit cartésien : simple ou généralisé.

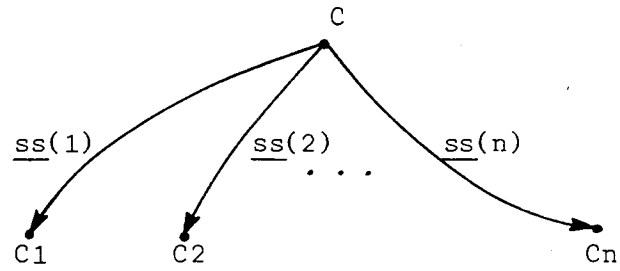
- Le produit cartésien simple -

Si C1, C2, ..., Cn sont des classes, une déclaration utilisant le produit cartésien simple s'écrit :

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

Cette définition crée une nouvelle classe, appelée C, dont les éléments sont formés de n composants tels que le $i^{ème}$ d'entre-eux appartient à la classe Ci.

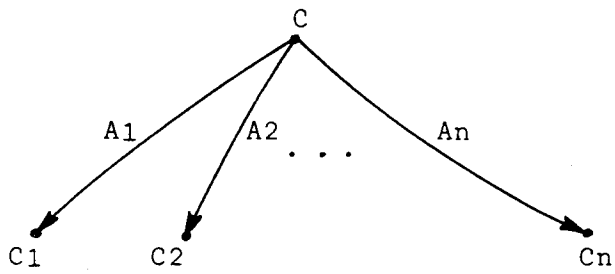
Dans le graphe, un nouveau sommet C est créé, ainsi que n arcs : le $i^{ème}$ d'entre eux va de C à Ci, et porte l'indication ss(i), pour signifier qu'il s'agit d'un sélecteur simple :



De plus, il est prévu une façon plus "évoluée" de définir des produits cartésiens simples, en associant un identificateur à chaque composant du produit :

def (C, cart (A1 ∈ C1, A2 ∈ C2, ..., An ∈ Cn))

Dans ce cas, il s'agit encore d'un sélecteur simple, mais les arcs sont représentés de la façon suivante :

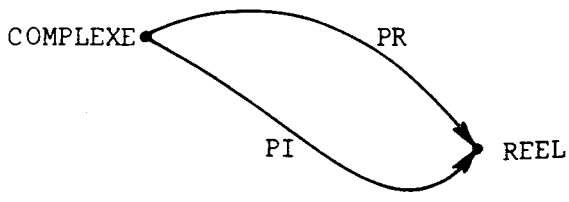


Si X est un objet qui appartient à une classe C définie par produit cartésien simple, alors le $i^{ème}$ composant de X est obtenu en écrivant s (i) (X), qui a pour résultat un élément de Ci. De plus, si l'identificateur Ai est associé avec ce composant, on obtient le même résultat en écrivant Ai (X).

Exemple :

def (COMPLEXE, cart (PR ∈ REEL, PI ∈ REEL))

le sous-graphe correspondant est :



Si Z est un complexe, on peut alors sélectionner sa "partie imaginaire" en écrivant :

PI (Z)

- Le produit cartésien généralisé -

On appelle également "séquence" le produit cartésien généralisé.

Si C1 est une classe et si C11, C12, ..., C1n sont n classes définies par :

def (C11, cart (C1))
def (C12, cart (C1, C1))
.
.
.
def (C1n, cart (C1, C1, ..., C1))

avec n occurrences de C1 dans le dernier produit cartésien, les éléments de la classe C1i ($1 \leq i \leq n$) sont appelés des séquences à composants dans C1, et de longueur i.

On peut alors définir une classe C2 de toutes les séquences à composants dans C1, et de longueur 1 à n :

def (C2, union (C11, C12, ..., C1n))

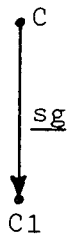
Cas général

On veut pouvoir définir la classe C de toutes les séquences à composants dans C1, y compris la séquence de longueur nulle et sans imposer de limite supérieure n pour la longueur de ces séquences. Une définition comme celle de C2 n'est donc pas suffisante, d'autant plus que cette définition ne produit pas, dans le graphe, de chemin allant de C2 à C1. Un tel chemin est évidemment indispensable pour représenter l'accès aux composants d'une séquence.

Ce sont les raisons pour lesquelles on introduit dans le mécanisme des classes, une façon particulière de construire la classe de toutes les séquences à composants dans C1 :

def (C, seq (C1))

Une telle définition crée un sommet C dans le graphe et un arc allant de C à C1. Cet arc est un arc de sélection générale :



Par exemple, la classe de tous les vecteurs à composants REELs est :

def (VECT, seq (REEL))

La classe de toutes les matrices REELles peut être définie par :

def (MAT, seq (VECT))

Ces définitions créent le sous-graphe :

Si X est un élément de la classe C des séquences à composants dans C1, un composant de X est obtenu en écrivant s (E) (X) où E est une expression. Si i est le résultat de E considéré comme une valeur entière non négative, et si la relation $i \leq n$ est satifaite où n est la longueur de la séquence X, le résultat de s (E) (X) et le $i^{ème}$ composant de X, qui appartient à la classe C1.

Par exemple, si M est une MATrice, on peut écrire :

s (3) (s (4) (M))

D'autre part, la fonction long (X) donne la longueur de la séquence X. Il faut noter qu'on ne peut pas déduire la longueur de X du simple fait que X est élément de C : la longueur d'une séquence X, élément de C, est une propriété de cette séquence particulière, et n'est pas une propriété attachée à la classe C. Cependant, il est également possible de définir une classe pour les séquences d'une longueur donnée : ce sont les séquences restreintes.

Cas restreint

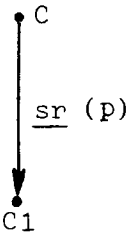
Si C1 est une classe, et E une expression, on peut définir :

def (C, cartn (C1, E))

Les éléments de C ont la même structure que les éléments de C2 défini par :

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

avec p occurrences de C1 dans le produit cartésien, où p est le résultat de E considéré comme une valeur entière non négative. Cependant, la représentation de C dans le graphe n'est pas celle d'un produit cartésien : la définition de C crée un nouveau sommet, et un arc de sélecteur restreint qui va de C à C1 :



Si X est un élément de C, un composant de X est obtenu en écrivant s (F) (X), où F est une expression dont le résultat i, considéré comme un entier, doit être tel que $1 \leq i \leq p$: le résultat de s (F) (X) est alors le $i^{ème}$ composant de X.

4.4.1.4. La définition du comportement dynamique

Les relations entre classes et les propriétés obtenues par les diverses formes de définition d'une classe sont d'une nature très primitive. En effet, on ne parvient à exprimer que deux sortes de propriétés ou relations :

- les éléments d'une classe C ont les mêmes propriétés que ceux d'une autre classe C' : c'est l'inclusion.
- les éléments d'une classe C sont construits avec des éléments d'autres classes C1, C2, ..., Cn : c'est le produit cartésien.

D'autres propriétés, qui se rapportent au comportement dynamique des éléments d'une classe, peuvent être définies sous la forme de relations fonctionnelles entre classes. Ce sont ces relations fonctionnelles qui donnent sa substance au mécanisme des classes : une fonction qui prend un argument dans la classe C1, pour donner un résultat dans une classe C2, est une propriété des éléments de la classe C1, car elle définit une "règle d'utilisation" des éléments de cette classe.

On distingue deux catégories de fonctions : les fonctions à appel implicite et les fonctions à appel explicite. Parmi les fonctions à appel implicite, il y a les prédicats et les conversions.

Quant aux fonctions à appel explicite, ce sont de simples procédures classiques. On peut aussi considérer que les sélecteurs définis par un produit cartésien sont des fonctions à appel explicite.

La différence entre ces deux catégories de fonctions se manifeste de la façon suivante : l'appel d'une fonction à appel explicite ne peut avoir lieu que s'il est écrit dans le programme et qu'il mentionne cette fonction particulière, tandis que l'appel des fonctions à appel implicite peut être provoqué automatiquement par les règles de la sémantique statique, ce qui n'interdit pas, si on connait un moyen d'identifier des fonctions, de les appeler aussi explicitement.

Dans ce qui suit, les définitions de toutes les fonctions se servent des lambdas du langage de base.

- Les prédicats -

Soient deux classes C1 et C2, définies au préalable, et telles qu'un arc d'inclusion va de C2 à C1.

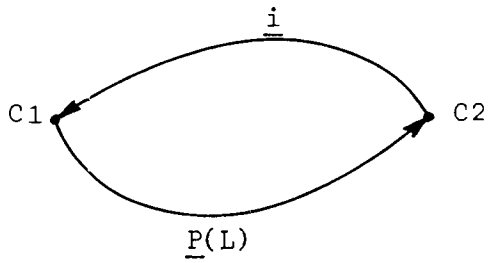
On peut alors définir la propriété caractéristique des éléments de C2 qui les distingue des autres éléments de C1. Ceci est réalisé par une déclaration de prédicat, qui a la forme :

def (I, pred (C1, C2, L))

où I est un identificateur - optionnel - et L un lambda à un paramètre,

et dont le résultat, au niveau du langage de base, sera considéré comme un bit . Lorsqu'il est appelé, un prédicat prend donc un paramètre, qui appartient à la classe C1, transmet sa valeur au lambda L, qui donne un résultat interprété comme vrai ou faux par la sémantique du langage : l'élément de la classe C1 utilisé comme paramètre est considéré comme étant aussi un élément de la classe C2 si et seulement si le résultat est vrai.

Dans le graphe, une telle définition crée un arc de prédicat, qui va en sens inverse de l'arc qui représente l'inclusion de C2 dans C1 :

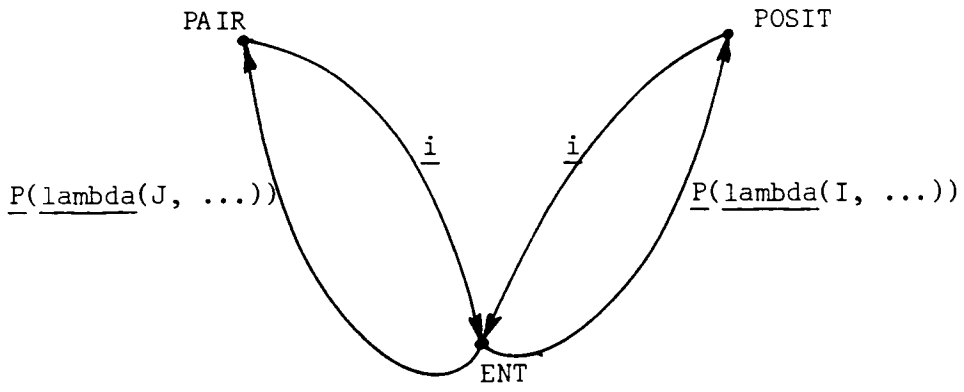


Cet arc indique le lambda L dont l'appel définit le prédicat proprement dit.

Exemples :

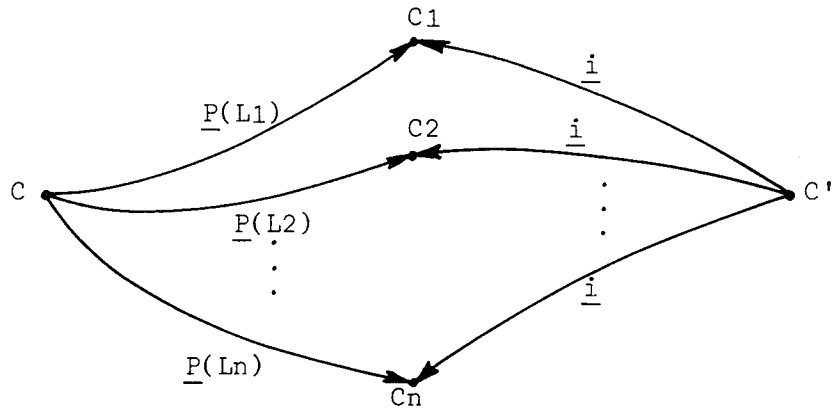
```
def (ESTPOS, pred (ENT, POSIT, lambda (I, supfix (I, 0))))
def (ESTPAIR, pred (ENT, PAIR, lambda (J, egfix (mod (J, 2), 0))))
```

Le sous-graphe correspondant est :

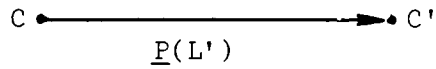


D'autre part, les prédicats existants et représentés dans le graphe, combinés avec les relations d'inclusion entre les classes, permettent de déduire d'autres prédicats.

Ainsi, si le graphe contient un sous-graphe de la forme générale suivante :

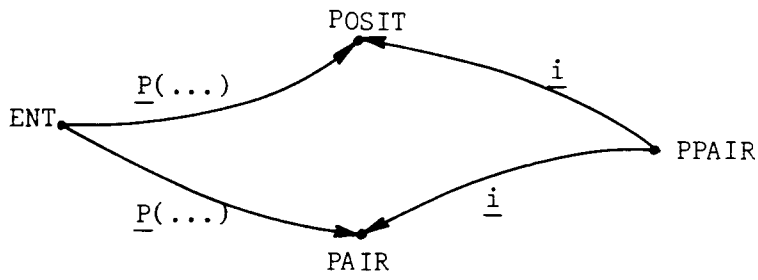


On peut déduire un nouvel arc :

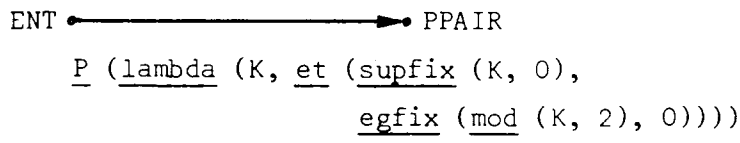


où le prédicat représenté réalise la conjonction des n autres prédicats.

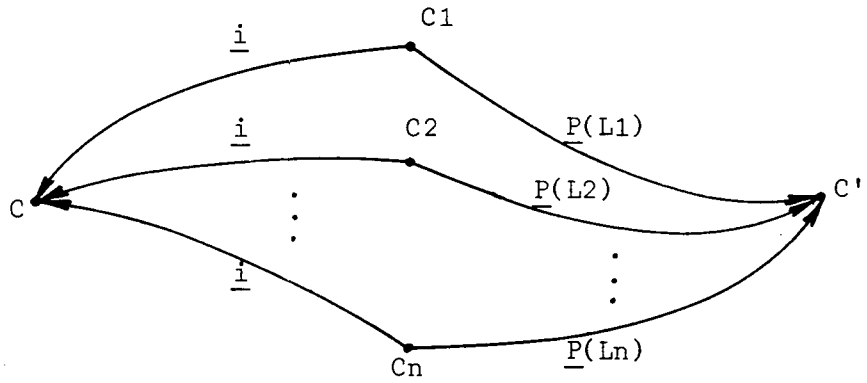
Par exemple, puisque l'on a le sous-graphe :



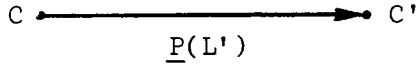
on peut déduire :



De même, si on a la configuration de sous-graphe :



On peut déduire :



où le prédicat représenté réalise la disjonction des n autres prédicats.

- Les conversions -

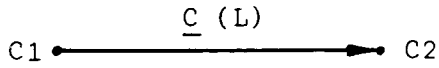
Soient deux classes C1 et C2. On définit une conversion automatique de C1 à C2 en écrivant :

def (I, conv (C1, C2, L)

où I est un identificateur - optionnel - et L un lambda à un seul paramètre.

Lorsqu'elle est appelée, une conversion prend donc un paramètre qui appartient à la classe C1, transmet sa valeur au lambda L, qui retourne un résultat qui est interprété comme étant un élément de la classe C2.

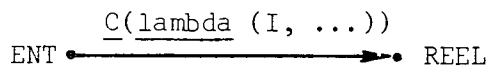
Dans le graphe, une telle définition crée un arc de conversion de C1 à C2, qui indique le lambda qui définit les actions de cette conversion :



Exemple :

def (, conv (ENT, REEL, lambda (I, fixflott (I))))

Ceci construit l'arc :



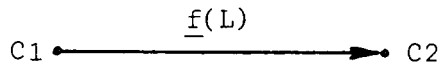
- Les procédures -

Soient deux classes C1 et C2. On définit une procédure à paramètres dans C1 et à résultat dans C2 en écrivant :

def (I, proc (C1, C2, L))

où I est un identificateur et L un lambda à un paramètre. Un appel de cette procédure exige un paramètre appartenant à la classe C1. La valeur de ce paramètre est transmise au lambda L qui retourne un résultat qui est pris comme étant un élément de la classe C2.

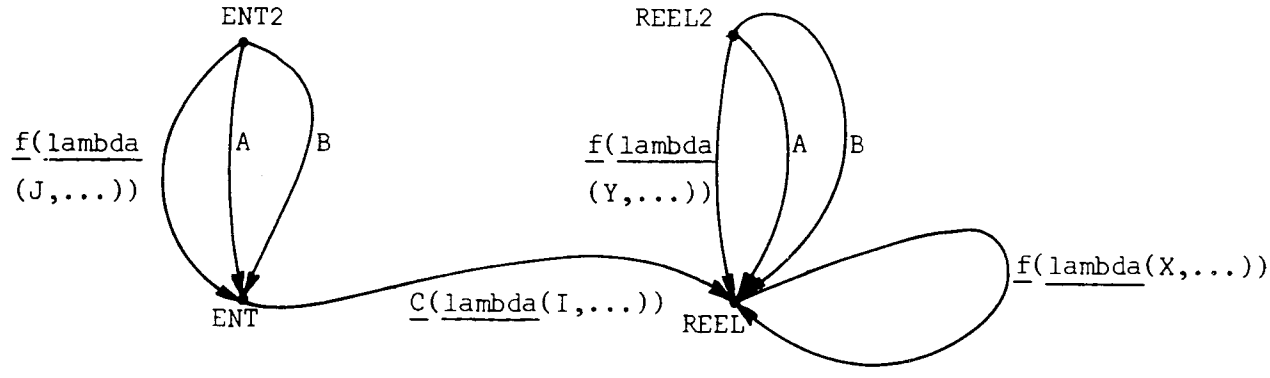
Dans le graphe, une telle définition crée un arc de fonction, de C1 à C2, qui indique le lambda qui constitue la procédure proprement dite :



Exemples :

```
def (RAC2, proc (REEL, REEL, lambda (X, ...)))
def (ENT2, cart (A ∈ ENT, B ∈ ENT))
def (REEL2, cart (A ∈ REEL, B ∈ REEL))
def (ADD, proc (ENT2, ENT, lambda (J, addfix (A (I), B (I)))))
def (ADD, proc (REEL2, REEL, lambda (Y, addflott (A (X), B (X)))))
```

Sous-graphe correspondant :



Cet exemple montre également la possibilité de définir des procédures génériques.

4.4.1.5. L'appartenance à une classe

Un objet primitif du langage de base n'appartient à aucune classe. Cependant, afin que le mécanisme des classes puisse remplir son rôle dans la sémantique statique, il est nécessaire de pouvoir dire qu'un objet X construit par une expression E du langage de base doit être considéré comme appartenant à une classe C.

Ceci est accompli par :

elem (C, E)

Le résultat de cette expression est l'objet primitif construit par E, considéré comme appartenant à C.
A ce résultat, une déclaration permet de donner un nom :

def (I, elem (C, E))

Exemples :

def (PI, elem (REEL, 3.1415926535))

Ceci a un effet analogue à la déclaration d'identité d'Algol 68 :

reel PI = 3.1415926535

Mais ici, on part d'une valeur qui, en elle-même, n'a pas de type (elle n'a qu'une représentation interne d'une certaine forme) et d'une classe, REEL, qui n'était pas imposée au départ par la définition du langage, car elle-même a été obtenue par déclaration.

De même, soit ALLOC un lambda à un paramètre et E une expression dont le résultat n sera considéré comme une valeur entière. On suppose alors que l'évaluation de :

eval (lier (ALLOC, E))

a pour résultat le premier emplacement libre de taille n à l'intérieur d'un autre emplacement géré par ALLOC à la façon d'une pile. Si l'on

déclare alors :

def (I, elem (ENT, copier (100, eval (lier (ALLOC, 1)))))

ceci a le même effet que :

ent I := 100

en Algol 68.
Il faut noter aussi que le résultat des conversions et des procédures appartient toujours à une classe déterminée.

4.4.1.6. L'utilisation du graphe

Etant donné le langage de base choisi et les définitions que l'on peut faire avec le mécanisme des classes, le seul contexte dont on peut dire qu'il "exige" un objet d'une classe déterminée est celui du paramètre d'un appel de procédure.

Soit dans A un objet d'une classe donnée. Sa valeur est un objet primitif A' du langage de base.

L'appel d'une relation fonctionnelle F avec A comme paramètre, qu'il s'agisse d'un prédicat, d'une conversion ou d'une procédure, est représenté dans le langage de base par l'évaluation d'un module obtenu en liant à A' l'unique paramètre du lambda L utilisé dans la définition de F :

eval (lier (L, A'))

Cette expression, qui est vide de toute information relative aux classes, est obtenue seulement après qu'ait été effectué un contrôle de classes : si F est représenté par un arc de prédicat, de conversion ou de procédure, A doit être un élément de la classe C représentée par le sommet situé à l'origine de cet arc, ou d'une classe C' telle que i^* (C', C) est vrai. (On rappelle que i^* est la fermeture transitive de la relation i (C1, C2) qui est vraie lorsqu'un arc d'inclusion va de C1 à C2).

Dans un programme, seul l'appel de procédure peut provoquer un contrôle de classe. L'appel d'une procédure P avec un paramètre A a la forme :

app (P, A)

Soit L le lambda de la définition de P et C la classe exigée pour tout paramètre d'un appel de P. Si A est un élément de C, ou de C' telle que i^* (C', C) est vrai, alors l'expression en langage de base est produite directement.

Sinon, soit Ca la classe à laquelle appartient A. Comme pour le mécanisme des propriétés additionnelles, on définit une fonction de passage associée à chaque arc :

- pour un arc de prédicat construit avec un lambda K, cette fonction est :

  lambda (X, cond ((eval (lier (K, X)) , X), erreur))

- pour un arc de conversion, c'est le lambda de cette conversion.

Soit Q l'ensemble des chemins allant de Ca à C et n'empruntant que des arcs d'inclusion, de prédicat et de conversion.

- S'il n'y a qu'un seul chemin dans Q, ou si tous les chemins dans Q ne diffèrent entre eux que par les arcs d'inclusion qu'ils contiennent, soit alors f1, f2, ..., fn la suite des fonctions de passage des arcs de prédicat ou de conversion de l'un de ces chemins.
  Dans ce cas, la fonction de passage f de Ca à C est définie et elle a la forme :

f = lambda (X, eval (lier (fn, eval (lier (fn-1, ... eval (lier (f1, X))...)))))

  L'appel app (P, A) est alors reconnu comme étant correct et il est traduit en langage de base sous la forme :

  eval (lier (L, eval (lier (f, A'))))

- Par contre, si Q est vide, l'appel est incorrect.

- Enfin, s'il y a dans Q au moins deux chemins qui diffèrent entre eux par au moins un arc de prédicat ou de conversion, il y a une ambiguïté dans le mécanisme de type qui a été construit.

Exemple :

```
def (K, elem (ENT, 1))
app (RAC2, K)
```

Cet appel est traduit en :

```
eval (lier (L1, eval (lier (L2, 1))))
```

où L1 est le lambda qui définit RAC2 et L2 est :

```
lambda (I, fixfloot (I))
```

c'est-à-dire le lambda qui définit dans le graphe la conversion d'ENT à REEL.

4.4.1.7. Remarques sur l'ambiguïté

Dans le mécanisme des propriétés additionnelles et dans celui des classes, on a mentionné l'éventualité d'ambiguïtés dans le système de conversions et de prédicats qui a pu être construit par des déclarations.

Ce problème se pose lorsque plusieurs fonctions de passage différentes vont d'un type t1 à un type t2.

Le mécanisme d'extension peut alors réagir de deux façons :

- ou bien il ne fait aucun choix, laisse l'ambiguïté, et peut alors signaler une erreur. C'est l'option qui a été retenue ici.
- ou bien, en vertu d'un certain nombre de critères, il élimine certaines des fonctions de passage pour n'en garder, si possible, qu'une seule.

La première réaction peut être considérée comme "sûre", bien que brutale. J'estime néanmoins que la deuxième réaction est plus à même de permettre une certaine "finesse" dans la définition de la sémantique statique. Cependant, il semble qu'un critère vraiment satisfaisant, contenant aussi peu d'arbitraire que possible, reste encore à découvrir, bien que plusieurs propositions aient déjà été faites :

- ne garder que la fonction de passage correspondant au chemin "le plus court" [Sc7o].

- prévoir, lors de leur définition, le rangement des prédicats et des conversions en diverses catégories et ne garder que la fonction de passage correspondant au chemin dont tous les prédicats et toutes les conversions appartiennent à une catégorie donnée imposée, par exemple, par la procédure dont l'appel a provoqué la recherche d'une fonction de passage : c'est le mécanisme des "niveaux", décrit en [Jo71], [Jo72] et [Be73], qui peut être vu comme une généralisation du concept de "force de contexte" introduit par Algol 68.
- attacher un coût à chaque arc, et ne garder que la fonction de passage correspondant au chemin de coût minimum : ce sont les "bias" de Alice et Mike Fisher dans [Fi73].

Aucune de ces solutions ne proposant de critère vraiement significatif, on a préféré se limiter ici à la réaction brutale qui assimile l'ambiguïté à une erreur. Mais ce problème demande à être étudié plus en profondeur, et une amorce de discussion sur ce sujet est présentée dans [Sc74].

4.4.2. SUGGESTIONS POUR UN MECANISME PLUS GENERAL

Le mécanisme primitif des classes permet de considérer sous un aspect formel certains points des propriétés additionnelles et évite l'utilisation d'un système prédéfini de modes hiérarchiques. Cependant, ce mécanisme n'aborde vraiment que la définition du comportement dynamique, car, en ce qui concerne la configuration statique, à part le produit cartésien, il n'ouvre pas d'autres possibilités que celles déjà prévues dans le langage de base. En effet, une classe définie sans que soient spécifiées des relations - inclusion ou produit cartésien - avec d'autres classes voit la représentation de ses objets définie seulement de façon implicite par les fonctions qui les acceptent comme arguments ou qui les produisent comme résultat. Il serait préférable que la capacité de définir de telles classes indépendantes puisse s'accompagner de la capacité de définir explicitement les notations qui correspondent aux objets de cette classe. Il deviendrait alors vraiment possible de définir beaucoup plus librement l'analogue des modes de base imposés par le mécanisme hiérarchique. De même, on peut envisager de généraliser le mécanisme des classes primitif en introduisant la possibilité de définir des constructeurs qui seraient considérés comme de véritables fonctions dans le domaine des classes, dont les paramètres seraient des classes ou des objets et qui donneraient comme résultat une nouvelle classe chaque fois qu'on les utilise avec des arguments différents.

Dans ce qui suit, on suggère donc une forme possible pour un tel mécanisme de classes généralisé, dont l'étude détaillée doit faire l'objet de travaux futurs. Les notations syntaxiques utilisées sont, pour la plupart, introduites directement par des exemples, et l'on ne cherche pas ici à entrer dans le détail de ce qui pourrait constituer un langage de base pour ce mécanisme.

4.4.2.1. Points de départ

Tous les objets existants ou potentiels appartiennent à une classe prédéfinie, appelée l'univers et notée symboliquement U. Ainsi, toutes les classes, sauf U, sont incluses dans une autre classe, qui est soit U, soit une classe déclarée.

Une déclaration de classe spécifie alors :

- le nom C de la classe déclarée ;
- la classe C1 dans laquelle C est incluse ;
- les propriétés (éventuelles) qui sont spécifiques aux éléments de C par rapport aux éléments de C1.

Ceci sera écrit sous la forme :

'C' ≡ C1 • propriétés

4.4.2.2. Forme générale des déclarations

La déclaration de classe ci-dessus n'est qu'un cas particulier d'une forme plus générale de déclarations qui est utilisée aussi bien pour les classes que pour les objets.

D'autre part, pour le fonctionnement du mécanisme des classes lui-même, on supposera la préexistence d'une classe, appelée TAG, directement incluse dans U , dont les éléments sont des identificateurs construits selon les conventions habituelles.

Une déclaration a la forme suivante :

declaration : identifiant lien identifié
identifiant : expression
identifié : expression
lien : $\underline{\equiv} \mid \underline{\in} \mid \underline{\supset}$

où l'expression qui constitue "l'identifiant" doit avoir un résultat appartenant à la classe TAG.

Dans une déclaration de la forme :

E1 ≡ E2

l'identificateur, résultat de E1, est défini comme une "constante" synonyme du résultat de E2. Il peut s'agir aussi bien d'une déclaration d'objet que d'une déclaration de classe.

Dans une déclaration de la forme :

$$E1 \in E2$$

l'identificateur, résultat de E1, est défini comme un "paramètre formel" qui peut être lié à des objets appartenant à la classe obtenue comme résultat de E2.

Dans une déclaration de la forme :

$$E1 \supset E2$$

l'identificateur, résultat de E1, est défini comme un "paramètre formel" qui peut être lié à des classes incluses dans la classe obtenue comme résultat de E2.

La liaison d'un paramètre s'effectue lors d'un appel de fonction ou lors de l'utilisation d'un constructeur de classe. Elle remplace le lien "$\in$" ou "$\supset$" par un lien "$\equiv$" et remplace l'expression qui constitue l'identifié par celle qui constitue le paramètre effectif.

- Remarque sur les déclarations et les objets de la classe TAG

Lorsqu'un identificateur est utilisé dans une expression il doit avoir fait l'objet d'une déclaration.

Donc, si l'on veut définir un nouvel identificateur, il faut utiliser une "notation" pour les objets de la classe TAG, de façon à ne pas provoquer la recherche d'une déclaration préalable de l'identificateur correspondant : on a choisi de représenter de telles notations sous la forme d'un identificateur entouré de deux apostrophes qui joueront donc un rôle du même genre que celui de "QUOTE" en LISP.

Exemples :

Si l'on déclare :

$$'I' \equiv 1$$

Ceci est une déclaration de l'identificateur I, défini ici comme une constante synonyme de 1. Par la suite on pourra par exemple écrire add(I, 2).

Si l'on déclare :

'J' ≡ 'K'

J ≡ 2

la première déclaration rend J synonyme de l'identificateur K, si bien que la deuxième déclaration signifie en fait :

'K' ≡ 2

4.4.2.3. Sélection généralisée et environnement

Une des bases du mécanisme suggéré ici est une généralisation de la notion de sélection, telle qu'elle existe par exemple en Algol 68, pour arriver à définir ce que l'on appelle l'environnement d'une expression.

En Algol 68, si on déclare :

struct (ent A, reel B) S = ... ;

on peut écrire :

A de S

Cette notation syntaxique, dans d'autres langages, aurait la forme :

S•A

C'est cette dernière forme que l'on adoptera ici pour exprimer la sélection généralisé, dont le rôle ne se limite plus à la simple dénomination d'un composant d'une structure.

En effet, en s'appuyant sur les principes de la sélection généralisée, la signification de S•A est : "Evaluer A dans l'environnement établi par S". Si bien que l'on est autorisé à écrire également, par exemple :

S•(A+B)

Ceci signifie : "Evaluer A+B dans l'environnement établi par S".
Comme on le verra dans des exemples donnés par la suite, il y a plusieurs façons de construire ainsi l'environnement d'une expression. En particulier :

- Le produit cartésien établit un environnement, de la même façon que les déclarations d'un bloc en Algol 60. Ainsi, écrire :

  cart (A1 ∈ C1, A2 ∈ C2, ..., An ∈ Cn) • E

  revient à demander l'évaluation de E dans un "bloc" qui déclarerait A1 de classe C1, A2 de classe C2, ..., An de classe Cn. En général, c'est une construction de ce genre qui sera utilisée pour définir A1, A2, ..., An comme paramètres formels utilisés dans E.

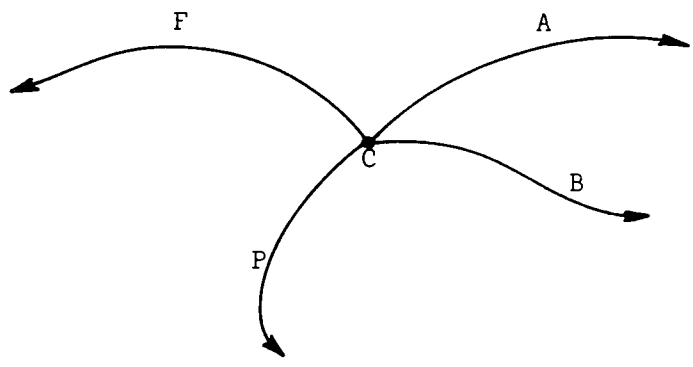
- Une classe établit un environnement. Ecrire :

  C•E

  donne accès, dans E, à tous les identificateurs associés aux propriétés qui ont été définies pour les objets de classe C. (L'utilisation du produit cartésien citée auparavant n'était d'ailleurs qu'un cas particulier de l'établissement d'un environnement par une classe).

D'une façon générale, si E1 et E2 sont des expressions, E1.E2 signifie l'évaluation de E2 dans l'environnement établi par E1.

Ici, comme précédemment, on fera usage du graphe des propriétés. Si le résultat X de E1 est de classe C , un certain nombre d'arcs partent du noeud qui représente C :



Ces arcs sont aussi bien des sélecteurs que des fonctions : tout ce qu'ils représentent constitue l'environnement dans lequel est écrit E2.

D'autre part, dans E2, l'objet X de classe C obtenu en résultat de E1 est conventionnellement nommé obj, c'est-à-dire que E1.obj a le même résultat que E1.

4.4.2.4. Inclusion d'une classe dans une autre

Comme pour le mécanisme primitif, l'inclusion peut être simple ou définie par le biais d'un prédicat.

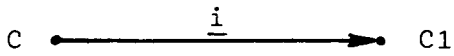
- Inclusion simple

L'inclusion simple ne sert qu'à "structurer" U ou une autre classe en un ensemble de sous-classes, sans rien apporter comme informations sur la nature ou les propriétés des éléments de ces sous-classes.
Si C1 est une classe, C est définie comme une nouvelle classe incluse dans C1 par :

$$C \equiv C1 \bullet \underline{inc}$$

Cette définition permet de considérer C comme disjointe de toutes les classes déjà incluses dans C1 par des déclarations analogues, et donne accès, à partir d'un objet appartenant à C, à l'environnement défini par C1.

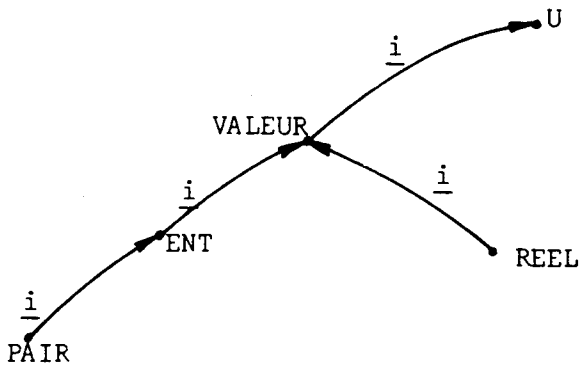
Effet sur le graphe :



Exemples :

'VALEUR' ≡ U•inc
'ENT' ≡ VALEUR•inc
'PAIR' ≡ ENT•inc
'REEL' ≡ VALEUR•inc

Graphe correspondant :



Le graphe construit par des inclusions simples est appelé l'arbre des inclusions. En effet, il n'y a plus, ici, les opérateurs union, inter, etc..., qui permettaient aussi de construire des inclusions simples. Les effets analogues à ce qu'ils permettaient d'accomplir sont obtenus ici au moyen de prédicat. Ceci, avec l'obligation de toujours préciser la classe contenante, a l'avantage d'éviter l'obtention de relations d'inclusion dont la signification était parfois difficile à déterminer avec le mécanisme primitif.

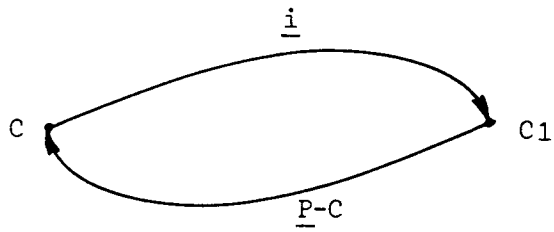
- Inclusion par prédicat

Une classe C est incluse par prédicat dans une classe C1 par la déclaration :

C ≡ C1•inc (E)

où E est une expression dont le résultat est interprété comme étant une valeur logique.

Dans le graphe, l'effet d'une telle déclaration est :

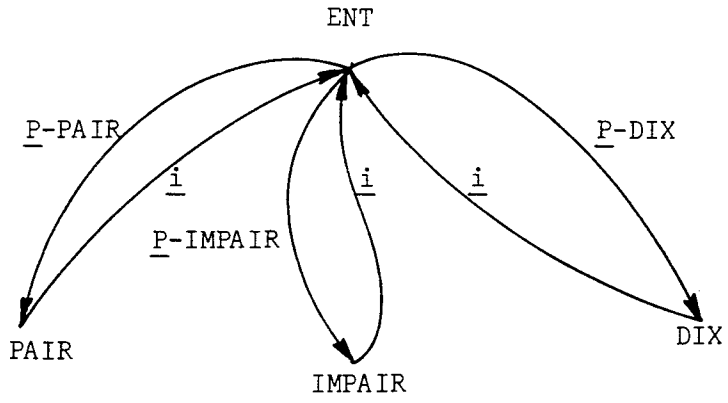


où P-C est un nom, automatiquement créé, pour pouvoir accéder explicitement au prédicat dans l'environnement de C1.

Exemples (on ignore la définition précédente de PAIR) :

'PAIR' ≡ ENT•inc (eg (mod (obj, 2), 0))
'IMPAIR' ≡ ENT•inc (non (P-PAIR (obj)))
'DIX' ≡ ENT•inc (et (infeg (1, obj), infeg (obj, 10)))

Graphe :



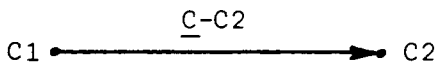
4.4.2.5. Conversion d'une classe dans une autre

On considère ici l'ensemble des conversions comme l'ensemble des significations d'une fonction générique, CONV, le choix d'une signification particulière dépendant de la calsse de l'argument fourni et de celle du résultat désiré.

On définit donc une conversion exactement comme une autre fonction (voir le paragraphe 4.4.2.10) :

CONV ≡ C1•fonc (C2, E)

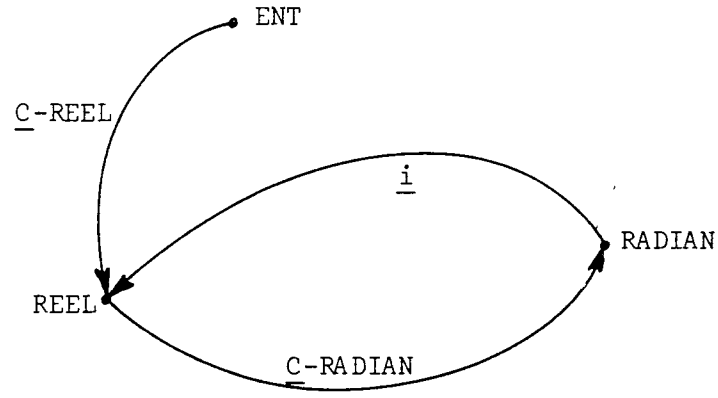
A la différence des autres déclarations de fonctions, ceci crée automatiquement un nom pour la conversion, qui est accessible dans l'environnement de C1 :



Exemples :

CONV ≡ ENT•fonc (REEL, fixflott (obj))
'RADIAN' ≡ REEL•inc
CONV ≡ REEL•fonc (RADIAN, ...)

Graphe :



4.4.2.6. Définition de notations

Un des rôles du langage de base doit être de permettre la construction de fonctions qui reconnaissent, au niveau lexicographique, une certaine catégorie de chaînes de caractères et qui lui font correspondre une certaine forme de représentation interne. Une telle fonction peut servir par exemple à associer un nombre binaire en virgule fixe à une suite de chiffres décimaux. Soit FIXE le nom de cette fonction, écrite en langage de base.

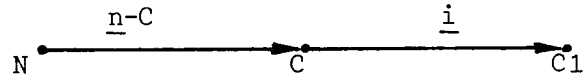
On peut considérer que chaque fonction de cette sorte définit un ensemble de notations au moyen d'un algorithme de reconnaissance. On donne alors à chaque ensemble de notations le nom de la fonction qui le définit. FIXE est donc un tel ensemble de notations. On pourrait définir de même l'ensemble FLOTT pour les nombres avec un point décimal.

Un ensemble de notations peut également être construit en indiquant simplement la liste de ses éléments. C'est un mécanisme qui existe déjà pour les types scalaires de Pascal [Wi73]. On considèrera dans ce cas que la représentation interne des objets correspondants est sans importance.

Soit donc N un ensemble de notations défini par l'une des deux méthodes ci-dessus. Pour définir une classe C, incluse dans une classe C1, et dont les objets ont des notations définies par N, on écrit :

C ≡ C1•nota N

Un ensemble de notation pouvant être assimilé à une classe de nature très élémentaire, ceci est représenté par :

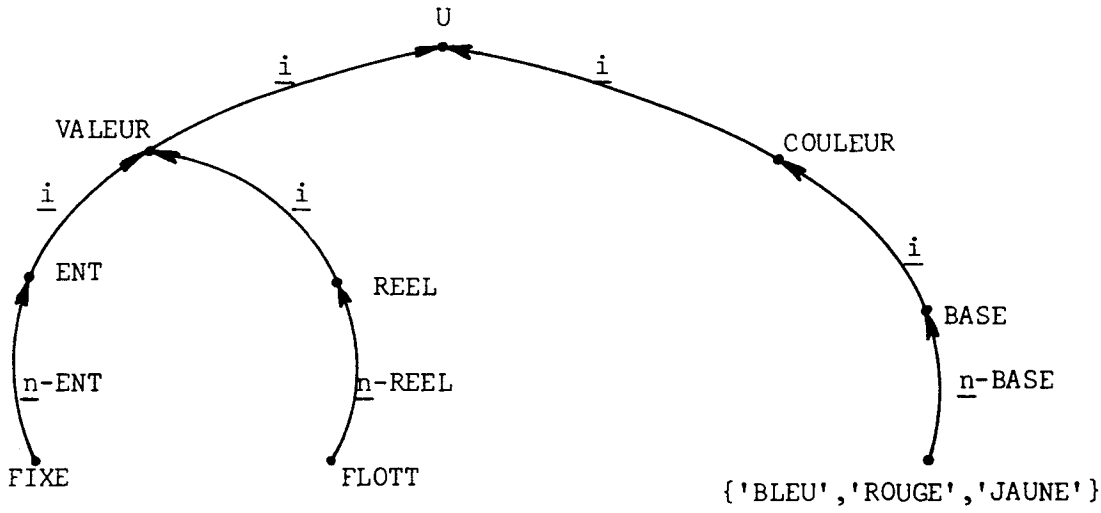


où n-C est une fonction de reconnaissance associée à N et qui produit, par convention, un résultat de classe C. Seuls des arcs de cette nature peuvent partir du noeud qui représente un ensemble de notations.

Exemples (on ignore les définitions précédentes de ENT et de REEL) :

'ENT' ≡ VALEUR•nota FIXE
'REEL' ≡ VALEUR•nota FLOTT
'COULEUR' ≡ U•inc
'BASE' ≡ COULEUR•nota {'BLEU', 'ROUGE', 'JAUNE'}

Graphe :



Ainsi, lorsque le contexte exige un objet de classe ENT, et que c'est la suite de caractères 123 qui est écrite dans ce contexte, cette suite de caractères est remplacée par n-ENT (123).

Remarque : ici, ainsi que dans le paragraphe suivant, on fait une utilisation du sybmole "•" qui peut constituer un léger "abus de langage" : Il est en effet utilisé pour n'indiquer que l'inclusion d'une classe dans une autre. Mais ceci rejoint sa signification première, car tout objet construit dans la classe contenue a accès à l'environnement établi par la classe contenante.

4.4.2.7. Produit cartésien

De même qu'un ensemble de notations permet de définir une classe en spécifiant qu'elle contient des objets atomiques d'une certaine forme, le produit cartésien permet de définir une classe en spécifiant qu'elle contient des objets non atomiques composés d'objets appartenant à d'autres classes.

Une classe C est incluse dans une classe C' et définie comme étant un produit cartésien à n composants par :

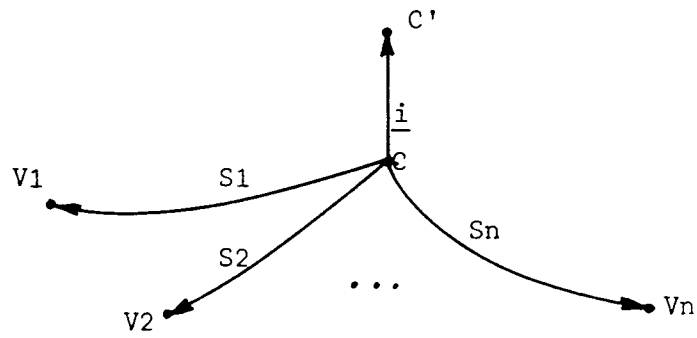
$$C \equiv C' \cdot \underline{cart}\ (D1, D2, \ldots, Dn)$$

où Di ($1 \leq i \leq n$) est une déclaration de constante ou de paramètre formel. Di a donc l'une des formes :

$$Ai \equiv Ei$$
$$\text{ou } Ai \in Ei$$
$$\text{ou } Ai \supset Ei$$

où Ai a pour résultat un TAG et Ei un objet ou une classe.

Une telle déclaration construit :



où Si est de la forme $\equiv$(Ai) ou $\in$(Ai) ou $\supset$(Ai) selon le lien utilisé dans Di, et où Vi est soit un noeud représentant une classe, soit un objet appartenant à une classe déterminée.

On entrevoit donc ici la possibilité d'élargir la nature des informations qui sont contenues dans le graphe des propriétés, en y introduisant aussi des objets. La question est d'ailleurs posée de savoir s'il faut conduire

cela beaucoup plus loin, jusqu'à considérer les classes elles-mêmes comme des objets à part entière. ECL [We70] a d'ailleurs déjà ouvert cette voie en considérant qu'il existe des valeurs de mode "mode".

D'autre part, une classe définie comme produit cartésien sert de générateur pour les objets qui lui appartiennent. En effet, si une classe C est définie par :

$$C \equiv C' \cdot \underline{cart}\ (D1, D2, \ldots, Dn)$$

où Di1, Di2, ..., Dip ($1 \leq ij \leq n$, $1 \leq j \leq p$) sont des déclarations de paramètres formels, un objet appartenant à C est généré par :

$$C\ (F1, F2, \ldots, Fp)$$

où Fj est une expression qui a pour résultat un objet susceptible d'être lié avec le paramètre formel défini par Dij.

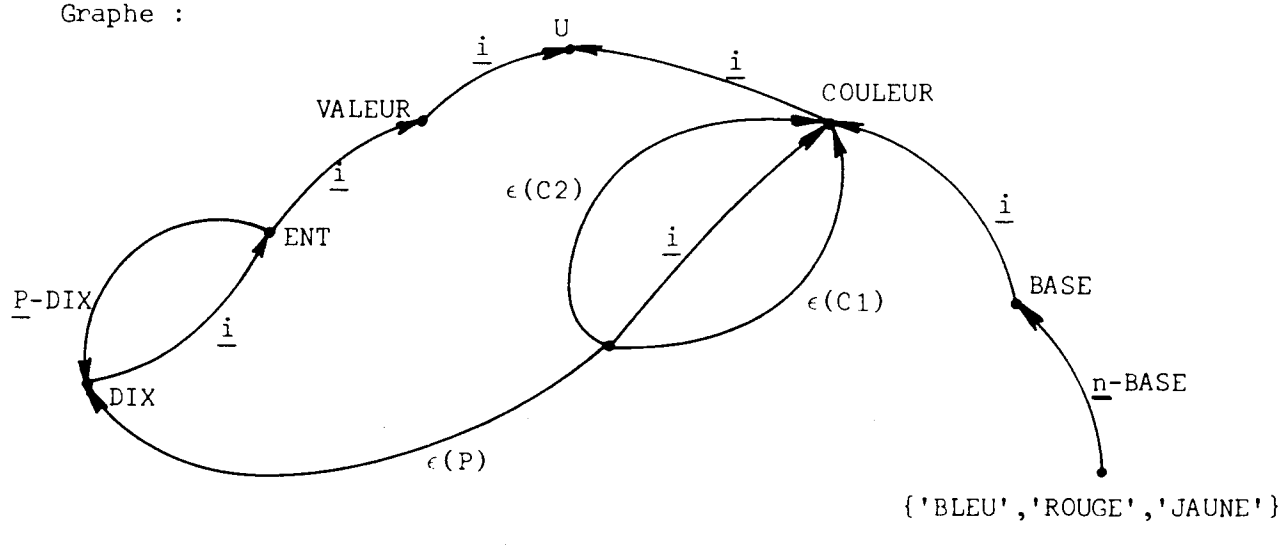
On peut également envisager un mécanisme de liaison partielle des paramètres qui permettrait d'obtenir non pas un objet appartenant à C mais une sous-classe de C.

Exemple :

```
'MELANGE' ≡ COULEUR·cart ('C1' ∈ COULEUR,
                          'C2' ∈ COULEUR,
                          'P' ∈ DIX)
```

où P sert à représenter la proportion de la quantité de couleur C1 par rapport à la quantité de couleur C2 utilisée dans le mélange.

Graphe :



Exemple de MELANGEs :

'ORANGE' ≡ MELANGE (ROUGE, JAUNE, 5)
'VERT' ≡ MELANGE (BLEU, JAUNE, 5)

On peut également avoir la sous-classe des verts, en faisant une liaison partielle de paramètres :

'VERTS' ≡ MELANGE (VERT)

Ceci a le même effet que :

'VERTS' ≡ MELANGE•cart ('C1' ≡ VERT, 'C2' ∈ COULEUR, 'p' ∈ DIX)

Si bien que l'on peut obtenir :

'TURQUOISE' ≡ VERTS (BLEU, 5)
'VERTJAUNE' ≡ VERTS (JAUNE, 5)

4.4.2.8. Séquences

Il est évidemment nécessaire de pouvoir définir des séquences. Des définitions de classes correspondantes ont été proposées dans le mécanisme des classes primitif, dont les possibilités semblent déjà assez générales.

Cependant, plusieurs questions restent encore insuffisamment explorées. En particulier :

- par quels moyens peut-on construire une séquence ?
- quelles sont les opérations d'itérations souhaitables pour permettre une utilisation simple des éléments d'une séquence ?

Il faudrait également prévoir que les séquences puissent servir de support à la définition de fonctions à nombre "variable" de paramètres. Ceci existe d'ailleurs en BASEL. Mais il faudrait aussi pouvoir construire, à l'aide de séquences, des constructeurs de classes à nombre variable de paramètres, comme struct ou tuple de BASEL. Ces problèmes seront étudiés dans le cadre de travaux ultérieurs.

4.4.2.9. Constructeurs de classes

Un constructeur de classe est analogue à ce que sont struct ou ref en Algol 68 : struct, par exemple, est utilisé pour construire des modes, comme

compl. Quand on déclare compl Z = ..., Z est un compl, c'est-à-dire un struct (reel PR, reel PI). Mais, considéré de façon plus "globale", Z est simplement un "struct", au même titre que tout autre objet construit avec struct. Ainsi, en reprenant la terminologie du mécanisme des classes, on est conduit à considérer que struct est lui-même une classe, dont struct(reel PR, reel PI) n'est qu'une sous-classe, construite par struct.

On considère donc ici qu'un constructeur de classe est lui-même une classe. Un constructeur de classe C, inclus dans une classe C', est défini par :

$$C \equiv C' \bullet \underline{\text{classe}}\ (D1, D2, \ldots, Dn) \bullet E$$

où

- Di ($1 \leq i \leq n$) est une déclaration de l'une des formes :

  $Ai \equiv Ei$
  ou $Ai \in Ei$
  ou $Ai \supset Ei$

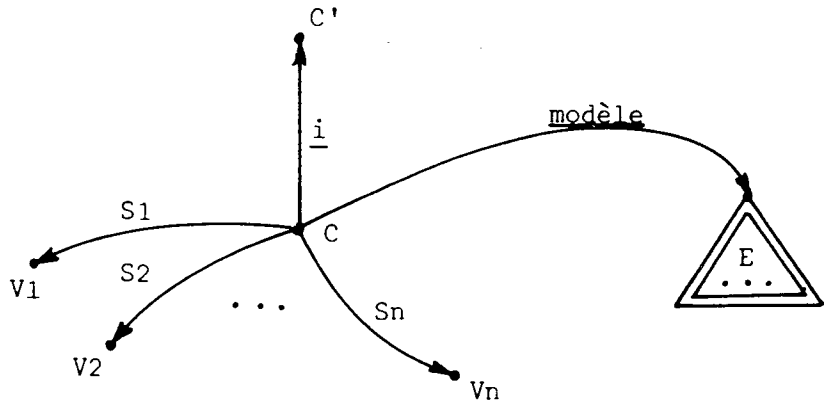
  Ai étant un TAG et Ei ayant pour résultat soit un objet soit une classe.

- E est l'une des façons déjà décrite de spécifier une classe :

  . inclusion
  . prédicat
  . notation
  . produit cartésien
  . séquence
  . constructeur de classes.

Parmi les Di, certaines déclarations définissent donc des paramètres formels : ce sont les paramètres du constructeur de classes.

La définition de C a l'effet suivant sur le graphe :



où Si est de la forme ≡(Ai), ∈(Ai) ou ⊃(Ai) selon le lien utilisé dans Di, et où Vi est soit un noeud représentant une classe, soit un objet. Di définit un paramètre formel lorsque le lien est ∈ ou ⊃.

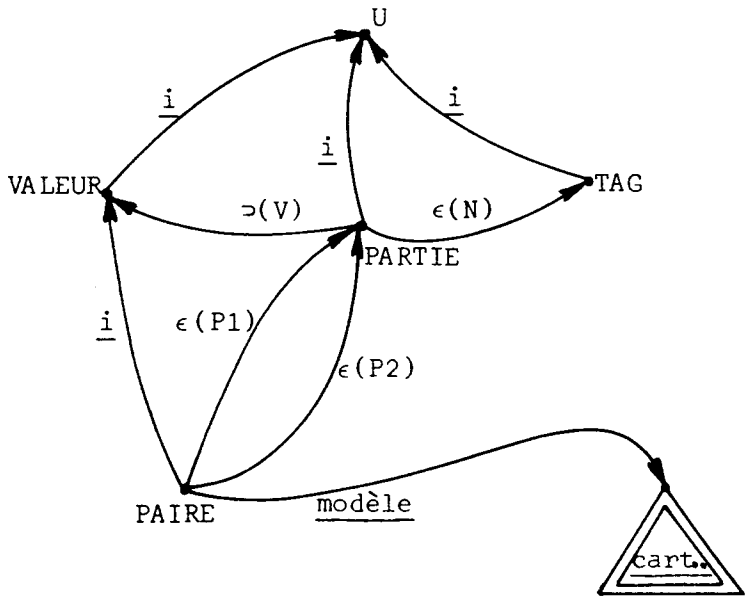
L'arc modèle sert à donner un accès, à partir de C, à une représentation conventionnelle de E. Cette représentation de E joue le rôle de prototype pour toutes les classes qui peuvent être construites en liant les paramètres de C. On peut d'ailleurs considérer qu'une construction de classe par C s'effectue à la façon d'une expansion de macro.

Exemple :

'PARTIE' ≡ U •cart ('N' ∈ TAG, 'V' ⊃ VALEUR)

'PAIRE' ≡ VALEUR•classe ('P1' ∈ PARTIE, 'P2' ∈ PARTIE)
•cart (P1•N ∈ P1•V, P2•N ∈ P2•V)

Graphe :



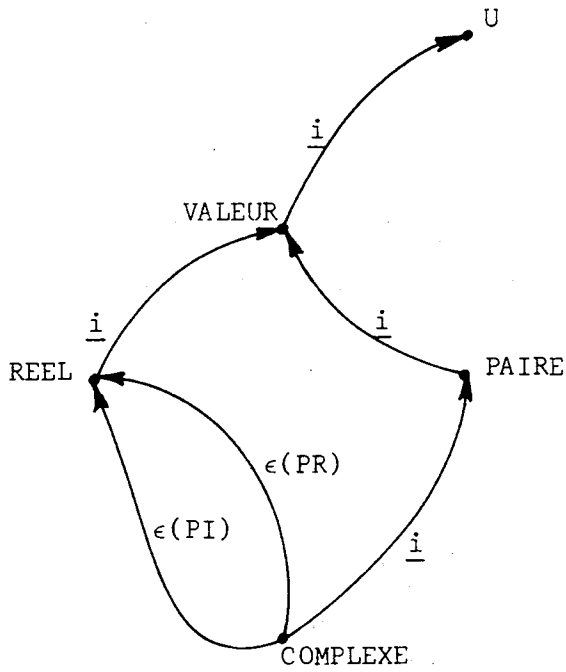
On peut alors écrire :

'COMPLEXE' ≡ PAIRE (PARTIE ('PR', REEL), PARTIE ('PI', REEL))

Ceci a le même effet que :

'COMPLEXE' ≡ PAIRE•cart ('PR' ∈ REEL, 'PI' ∈ REEL)

Effet sur le graphe :



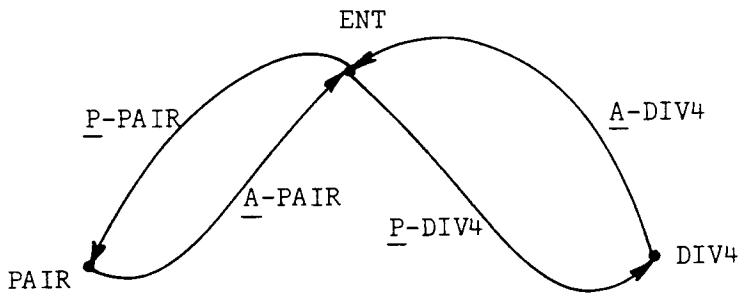
On dispose alors d'un générateur de COMPLEXEs :

'Z' ≡ COMPLEXE (3.14159, 2.71828)

Z est alors un COMPLEXE, donc aussi une PAIRE.

Remarque : Dans [Sc74] il est également suggéré que lorsqu'on lie les paramètres d'un constructeur de classes (appelé "general class" dans [Sc74]), ce n'est pas une inclusion simple que l'on obtient, mais une inclusion par prédicat qui, du constructeur vers la classe construite, permet de vérifier les valeurs des paramètres et qui, dans l'autre sens, se comporte comme une assertion sur ces valeurs. Il semble en effet que la présence de telles informations dans le graphe soit indispensable pour traiter certains problèmes de la sémantique statique, comme par exemple la comparaison de deux classes construites par un même constructeur. De

même, lors d'une définition de classe au moyen d'une inclusion par prédicat, il est suggéré dans [Sc74] d'avoir dans un sens un arc de prédicat et dans l'autre un arc d'assertion. Ainsi (exemple mentionné par S. Schuman [Sc74a]) si la classe PAIR et la classe DIV4 des entiers divisibles par 4 sont toutes les deux incluses dans ENT, on peut facilement découvrir qu'un élément de DIV4 peut toujours être utilisé dans un contexte qui exige un élément de PAIR :



4.4.2.10. Fonctions

Une fonction peut être simple ou générique, et lorsqu'elle est générique, elle peut être explicitement générique ou implicitement générique.

- Fonctions simples

Une fonction simple F à paramètre dans C1 et à résultat dans C2 est définie par :

'F' ≡ C1•fonc (C2, E)

Exemple :

'CONJ' ≡ COMPLEXE•fonc (COMPLEXE, COMPLEXE (PR, moins (PI)))

- Fonctions explicitement génériques

Une fonction F est explicitement générique lorsque le même TAG 'F' a été déclaré plusieurs fois comme fonction avec des paramètres de diverses classes, et que plusieurs de ces déclarations sont accessibles dans un même environnement.

Exemple :

```
'ADD' = cart (A ∈ ENT, B ∈ ent)•fonc (ENT, fixadd (A, B))
'ADD' = cart (A ∈ REEL, B ∈ REEL)•fonc (REEL, flottadd (A, B))
'ADD' = cart (A ∈ COMPLEXE, B ∈ COMPLEXE)•
                    fonc (COMPLEXE, COMPLEXE (ADD (A•PR, B•PR)
                                              ADD (A•PI, B•PI)))
```

Le choix d'une signification particulière est fait en fonction de la classe du paramètre effectif d'un appel. Dans [Sc74] il est également suggéré que la classe exigée par le contexte pour le résultat de l'appel soit aussi prise en compte dans le choix d'une signification.

- Fonctions implicitement génériques

C'est l'un des points les plus originaux du mécanisme des classes généralisé [Jo74] [Sc74]. Une fonction est dite implicitement générique quand la classe C1 de son paramètre est un constructeur de classes : une telle fonction sert de prototype pour autant de fonctions, ayant le même nom qu'elle, qu'il y aura de classes construites par le constructeur de classe considéré.
(Si l'on envisage alors ce prototype comme le corps d'une définition de macro, les paramètres qui permettent de l'"adapter" au cas particulier de chaque classe construite sont ceux du constructeur de classes).

Exemple :

```
'DIST' ≡ PAIRE•fonc (REEL, SQRT (ADD (puiss (P1•N, 2),
                                      puiss (P2•N, 2))))
```

La fonction DIST est implicitement générique.

Si, en plus de COMPLEXE, on déclare également :

'VECT' ≡ PAIRE (PARTIE ('X', ENT), PARTIE ('Y', ENT))

on obtient automatiquement deux significations pour DIST :

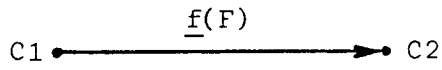
COMPLEXE•fonc (REEL, SQRT (ADD (puiss (PR, 2), puiss (PI, 2))))
et VECT•fonc (REEL, SQRT (ADD (puiss (X, 2), puiss (Y, 2))))

- Effets sur le graphe

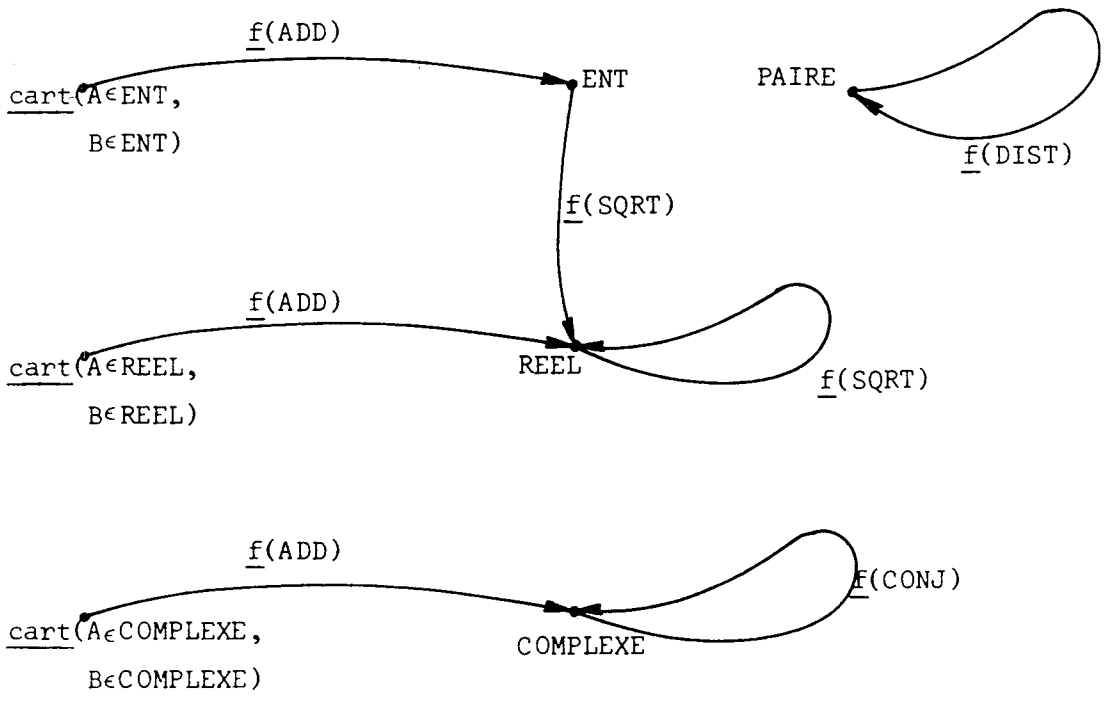
Toute définition de fonction crée un arc dans le graphe.

F ≡ C1•fonc (C2, E)

crée :

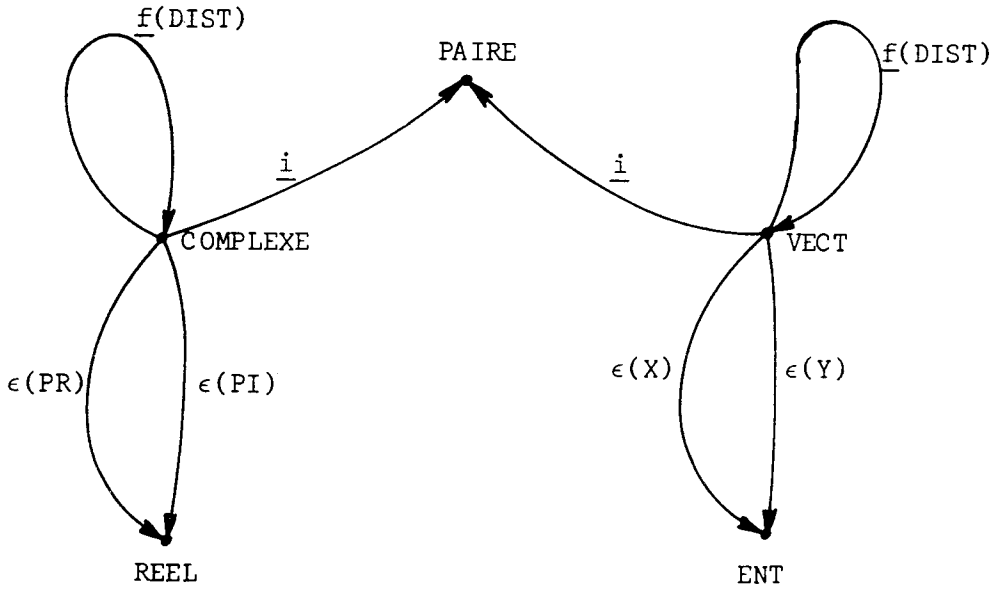


Exemples :



D'autre part, à toute classe obtenue par utilisation d'un constructeur correspondent autant d'arcs de fonction partant du noeud qui représente cette classe qu'il y a de fonctions implicitement génériques partant du constructeur.

Exemple :



4.4.2.11. Utilisation du graphe dans la sémantique statique

Le principe général d'utilisation du graphe est le même ici que pour le mécanisme primitif des classes : lorsqu'un objet de classe C1 est obtenu dans un contexte qui exige un objet de classe C2, on recherche l'existence d'une fonction de passage de C1 à C2.

Le problème de l'ambiguïté se pose d'une façon similaire à ce qui a déjà été dit.

Toutefois, l'exemple suivant permet de suggérer que le critère du chemin le plus court n'est pas si "naïf" qu'il parait, car il semble pouvoir donner souvent le résultat que l'on recherche :

'MILE' ≡ VALEUR•inc

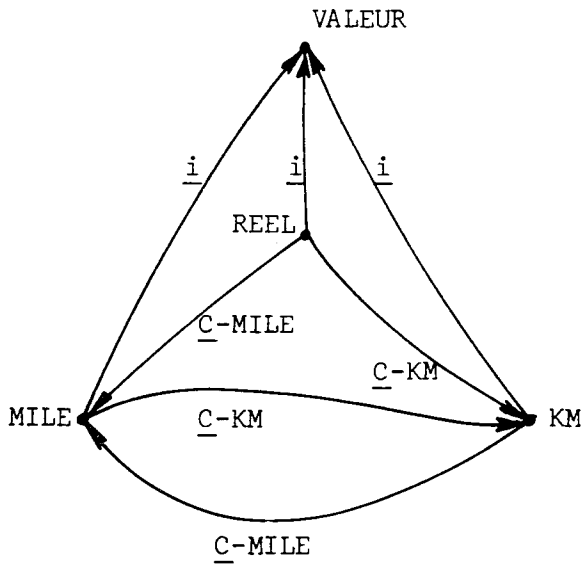
'KM' ≡ VALEUR•inc

CONV ≡ REEL•fonc (MILE, obj)

CONV ≡ REEL•fonc (KM, obj)

CONV ≡ MILE•fonc (KM, prod (obj, 1.610))

CONV ≡ KM•fonc (MILE, div (obj, 1.610))



Ainsi, un REEL peut toujours être utilisé comme un MILE ou comme un KM : il subit une conversion (C-MILE ou C-KM de l'environnement de la classe REEL), mais cette conversion n'affecte pas sa valeur.

D'autre part, lorsqu'un KM est utilisé dans un contexte qui exige un MILE, il est converti en MILE, et vice-versa.

Cependant, cette définition est toujours incomplète car, si les opérations classiques d'addition, soustraction, ... sont définies sur les REELs, elles ne le sont ni sur les MILEs ni sur les KMs.

On a alors plusieurs options :

- soit on les redéfinit toutes pour MILE et toutes pour KM. Cette solution est évidemment correcte, mais déraisonnable si l'on considère toutes les réécritures de choses identiques que cela implique.

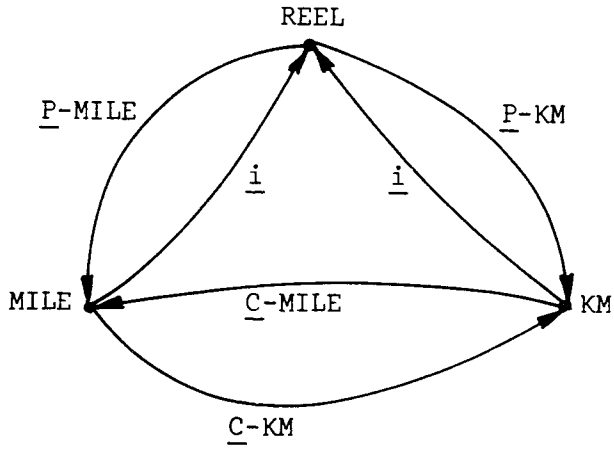
- soit on définit deux autres conversions :

CONV ≡ MILE•fonc (REEL, obj)
CONV ≡ KM•fonc (REEL, obj)

On rencontre alors le même problème que si l'on avait déclaré :

'MILE' ≡ REEL•inc (vrai)
'KM' ≡ REEL•inc (vrai)
CONV ≡ MILE•fonc (KM, prod (obj, 1.610))
CONV ≡ KM•fonc (MILE, div (obj, 1.610))

Ce qui construit le graphe :



Ici, l'inclusion de MILE et de KM par prédicat identiquement vrai dans REEL est nécessaire pour pouvoir utiliser directement des MILEs ou des KMs de la même façon que des REELs : si A et B sont des KMs, on peut alors les additionner en utilisant directement l'addition sur les REELs.

Mais apparait alors un nouveau problème : rien n'interdit plus d'additionner des MILEs avec des KMs, sans faire subir à l'un des deux opérandes une conversion qui en ferait un objet de la classe de l'autre opérande.

Une solution est peut être à rechercher dans la direction suggérée par la notion de "type abstrait" introduite dans [Sc74].

5. CONCLUSION

-:-:-:-:-:-:-

Les principaux aspects du problème des langages extensibles viennent d'être présentés, certains avec beaucoup de détails, d'autres de façon plus schématique, selon l'importance ou l'avancement de ma participation aux travaux s'y rapportant.

Dans l'état actuel de ce qui a été fait dans ce domaine, il me semble que l'on peut envisager, dès maintenant, la conception puis l'implémentation d'un langage extensible complet dont le fonctionnement, à la différence d'autres expériences déjà faites, sera entièrement gouverné par l'organisation de sa définition formelle.

Il convient donc de définir un langage de base, selon les directions indiquées à la fin du chapitre 3 et d'en réaliser une implémentation dont l'architecture soit conçue pour accueillir l'extensibilité de la sémantique dynamique au moyen de l'interpréteur extensible. Il faut aussi choisir, comme cela a été indiqué au chapitre 2, un mécanisme d'extension syntaxique qui, tout en offrant plus de possibilités que celui des déclarations de priorités d'Algol 68, n'atteigne pas la puissance encombrante des macros syntaxiques générales. Il faut enfin mener à leur terme les réflexions sur l'extensibilité de la sémantique statique en prenant comme support le mécanisme des classes généralisé suggéré au chapitre 4. En regroupant les mécanismes de la syntaxe et de la sémantique statique, on obtient alors un véritable traducteur extensible.

Ainsi, le traducteur et l'interpréteur étant tous les deux modifiables au moyen d'extensions, on a bien le moyen de définir complètement des langages et d'obtenir en même temps leur implémentation. C'est une étape importante à laquelle il faut arriver.

Cependant, il ne faut pas perdre de vue le contexte plus large dans lequel s'inscrit ce travail, et dont l'envergure dépasse le cadre classique des langages de programmation, extensibles ou non. En effet, les travaux de recherche sur les langages vus comme moyen d'accès à un ordinateur doivent aujourd'hui se situer par rapport à deux courants qui concernent la façon sont ces langages seront utilisés. Il s'agit, d'une part, des réflexions actuelles sur la méthodologie de la programmation et, d'autre part, des travaux récents sur les langages "de très haut niveau" [Le74].

A partir de 1968 il est peu à peu apparu que les programmes que l'on écrivait, qu'il s'agisse d'applications de toutes sortes, de compilateurs ou de systèmes d'exploitation, devenaient de plus en plus complexes, atteignaient des tailles de plus en plus grandes, dépassaient déjà de loin ce qu'un être humain peut maîtriser à lui seul. Il devenait nécessaire de mettre au point des méthodes et des techniques qui permettent d'établir un certain nombre de conventions dans la façon de construire un programme, qui favorisent le respect de ces conventions, qui rendent possible un contrôle global de la structure du programme et qui donnent à son sujet l'assurance d'un fonctionnement correct. De ces considérations est issue la "programmation structurée", dont Dijkstra fut l'un des pionniers [Di68, Di69], et qui est maintenant reconnue comme une discipline fondamentale pour le développement des applications et des systèmes à venir [Da72]. Mais pour satisfaire aux exigences de cette discipline, un simple langage algorithmique ne suffit plus. C'est ainsi qu'est apparue l'idée de système de programmation, dont l'objectif et de donner au programmeur la possibilité de se servir d'un langage de programmation, tout en mettant à sa disposition un ensemble de mécanismes qui l'assistent dans la conception de son programme, qui l'aident à en rendre l'écriture systématique et qui l'incitent à appliquer les principes de la programmation structurée. Par exemple, certaines des idées de base du mécanisme de modules proposé dans [Sc74a] peuvent fort bien servir de point de départ pour la conception d'un tel système.

Mais, au sein d'un système de programmation, la nature du langage dont on dispose est déterminante pour l'aisance avec laquelle peuvent être mis en pratique les principes de la programmation structurée. Dans ce sens, il est maintenant clair que les techniques des langages extensibles ont un rôle privilégié à jouer. En effet, un langage extensible met en oeuvre un principe fondamental de la programmation structurée, celui des "niveaux d'abstraction", qui consiste à définir par couches successives [Wi71] les objets et les opérations dont on a besoin : un objet ou une opération sera défini en terme d'objets ou d'opérations plus élémentaires, eux-mêmes construits avec des objets ou opérations encore plus élémentaires, etc..., jusqu'au niveau le plus bas du langage qui donne directement accès aux représentations sur la machine. On voit donc que, pour cette façon structurée d'envisager l'organisation des programmes, les mécanismes d'exten-

sion syntaxiques et sémantiques sont actuellement les outils les mieux adaptés. Cependant, si l'on veut aussi pouvoir opérer de façon "descendante" pour ces définitions par couches, comme cela est préconisé, il faut alors des outils supplémentaires qui, eux, se situent dans le système de programmation et non plus dans le langage lui-même. On doit en effet pouvoir utiliser des objets, des opérations ou même des constructions syntaxiques dans un programme, et contrôler que ce programme est statiquement correct, sans être forcé de préciser d'abord jusque dans les moindres détails les couches successives qui définissent ces objets, ces opérations ou ces constructions syntaxiques.

Les travaux dans ce domaine, en relation avec ceux sur les langages extensibles, sont sans aucun doute susceptibles de transformer profondément, dans un avenir assez proche, toute l'activité de programmation.

L'autre tendance dont il faut aussi tenir compte pour les recherches actuelles dans le domaine des langages est celle des langages "de très haut niveau". Cette tendance, comme les efforts antérieurs qui ont amené à la conception des langages de programmation classiques, a pour but de rapprocher encore le langage des processus de pensée du programmeur d'applications, en allant jusqu'à rendre la machine complètement "invisible". En fait, les langages extensibles, eux aussi, ont cet objectif, et l'on peut d'ailleurs considérer (voir certains articles dans [Le74], en particulier [Li74]) qu'ils ont "lancé" le courant des langages de très haut niveau et que la façon dont ils sont conçus aujourd'hui s'inscrit même très bien dans ce courant. Cependant, à mon avis, cette tendance vise encore plus loin que les langages extensibles, et c'est pour cela qu'il est important non seulement d'en tenir compte mais aussi d'y participer.

En effet, si on regarde de près les langages classiques de haut niveau, et même les langages extensibles, malgrè les abstractions que ceux-ci permettent de construire, on constate qu'ils sont tous empreints des caractéristiques les plus élémentaires des machines, de leur structure et de leur fonctionnement. On retrouve toujours dans ces langages des structures de données fortement marquées par la nature linéaire de la mémoire ; on y retrouve, plus ou moins bien camouflées, les notions d'adresses et de pointeurs ; on y retrouve le règne, bien souvent arbi-

traire, de l'exécution séquentielle ; on y retrouve, de façon explicite, des distinctions entres les diverses formes de codification binaire des informations, etc.... C'est pourquoi certains groupes de recherche ont commencé à penser qu'il faudrait franchir une nouvelle étape pour s'éloigner des contraintes de la machine encore plus que ne le permettent les langages classiques de haut niveau ou même les langages extensibles. Ce n'est que vers 1972/73 que ce courant de travaux a pris un nom et ce furent les langages "de très haut niveau".

Les efforts dans ce domaine portent actuellement sur plusieurs points fondamentaux. Ainsi, on a cherché d'abord à éliminer les contraintes de l'exécution séquentielle explicite lorsque celle-ci est inutile. Ceci a pour conséquence non seulement de permettre l'écriture de certains ensembles d'instructions sans tenir compte de l'ordre dans lequel elles seront exécutées, mais encore d'introduire des mécanismes de programmation non déterministe et aussi d'éliminer certaines structures de données de nature séquentielle, comme les tableaux. On est alors amené à introduire de nouvelles sortes d'opérations et de nouvelles formes d'agrégats dans ces langages, et les ensembles, au sens de la théorie des ensembles, sont actuellement considérés comme d'excellents candidats, avec toutes les opérations que l'on peut imaginer pour les construire et les manipuler. Plusieurs langages expérimentaux, de nature toujours algorithmique, ont déjà été construits autour de ces idées, notamment SETL par Schwartz [Sz73] et certains travaux récents de Earley [Ea73, Ea74].

D'autre part, si un langage permet de construire et de manipuler des ensembles, la voie est ouverte aux mécanismes d'accès associatif , c'est-à-dire que l'on peut y introduire des mécanismes pour accéder à un objet en spécifiant ses propriétés caractéristiques ou certaines de ses relations avec d'autres objets. On rejoint alors directement des langages comme LEAP [Fe69, Fe72] (voir aussi l'étude récente de Bobrow dans [Bw74]), ou encore les langages d'interrogation de bases de données relationnelles comme le langage ALPHA de Codd [Cd71, Cd74], et même les systèmes de questions-réponses du genre de celui développé par Colmerauer [Cl72], et qui fait appel à des techniques de démonstration automatique que l'on peut considérer là comme généralisations des mécanismes d'accès associatif.

On est alors tenté de faire le pas vers l'intelligence artificielle, car on entrevoit déjà que l'étape suivante sera de concevoir un environnement pour accueillir des langages d'application qui vont encore plus au devant de l'utilisateur et de son problème. En effet, rien n'interdit de penser que pour bien des applications, la notion d'algorithme elle-même peut-être éliminée. Au lieu de langages qui, comme les langages algorithmiques que l'on connait aujourd'hui, ne permettent que de décrire une façon particulière de construire les résultats d'un problème, on peut commencer à imaginer des langages non algorithmiques qui permettront d'abord de définir les données - pas seulement numériques - d'un problème et les relations - logiques ou algébriques - de ces données entre elles, puis de poser des questions et d'établir un dialogue avec la machine au sujet du domaine de travail ainsi construit. Comme cela a déjà été dit ailleurs, ce serait "quoi" au lieu de "comment". C'est certainement, à mon avis, le genre de travaux à poursuivre si l'on veut ouvrir de plus en plus à des non-informaticiens l'accès aux services que peut leur rendre un ordinateur.

Dans l'état actuel des travaux sur les langages on distingue donc deux façons de communiquer avec une machine, selon qu'on formule un algorithme ou un problème. Dans le premier cas, il s'agit d'une activité de programmation, et les techniques qu'il convient de développer aujourd'hui sont celles de l'extensibilité et des systèmes de programmation, afin de mettre en oeuvre des outils pratiques et sûrs pour construire des programmes. Dans le deuxième cas, il s'agit d'une activité de dialogue avec une machine capable d'assimiler, et d'organiser elle-même, un certain nombre de faits, puis d'opérer les déductions qui l'amèneront à découvrir les réponses aux questions posées sur ces faits. Pour que cela soit possible, il convient de développer un langage non algorithmique pour exprimer ce dialogue, de concevoir et d'implémenter les mécanismes d'analyse et de génération syntaxiques et sémantiques qui lui sont associés, et de construire les processus de déduction et de calcul qui donneront à la machine la faculté de dialoguer "intelligemment".

REFERENCES

-:-:-:-:-:-

[Al68a] - ALBER, K., OLIVA, P., and URSCHLER, G.
Concrete Syntax of PL/1
IBM Laboratory Vienna
June 1968, No. TR 25.084

[Al68b] - ALBER, K., and OLIVA, P.
Translation of PL/1 into abstract text
IBM Laboratory Vienna
June 1968, No. TR 25.086

[Ba73] - BAJAR, V.
Etude de la compilation de BASEL, langage de la famille d'Algol 68
Thèse de troisième cycle, U.S.M. Grenoble
Février 1973

[Be72] - BERT, D.
Notes non publiées

[Be73] - BERT, D.
Etude d'éléments fondamentaux des langages de programmation
Thèse de 3ème cycle, U.S.M. Grenoble
Mai 1973

[Be74] - BERT, D.
Analyseur de double grammaire par macros syntaxiques
Rapport scientifique CNRS
Janvier 1974

[Bk72] - BACKUS, J.
Reduction languages and variable-free programming
IBM Research Laboratory, San Jose
April 1972, No. RJ 1010

[Bo66] - BOUSSARD, J.C., COHEN, J., et JORRAND, Ph.
Etude et réalisation d'un langage d'édition
5ème Congrès de l'AFIRO, Lille
Juin 1966, pp. 456-459

[Bo67] - BOUSSARD, J.C., COHEN, J., JORRAND, Ph. et TRILLING, L.
Paramétrisation des compilateurs
6ème Congrès de l'AFIRO, Nancy
Mai 1967, Section 2, pp. 144-153

[Bo71] - BOUSSARD, J.C., et DUBY, J.J. (Eds)
Rapport d'évaluation Algol 68
RIRO, Revue Bleue
Février 1971, No. B-1, pp.15-106

[Bw74] - BOBROW, D.G., and BERTRAM R.
New programming languages for artificial intelligence research
Computing Surveys
September 1974, Vol.6, No.3, pp.153-174

[Br71] - BROSGOL, B.M.
An Implementation of ECL data types
International symposium on extensible languages
December 1971, SIGPLAN Notices, Vol.6, No.12, pp.87-95

[By7oa] - BERRY, D.M.
Property grammars, Basel, Contour Model and Coercion
Brown University Report
May 1970

[By7ob] - BERRY, D.M.
Introduction to Oregano
Brown University Report
September 1970

[By74] - BERRY, D.M.
On the design and specification of the programming language Oregano
Computer Science Department, UCLA
January 1974, No. UCLA-ENG-7388

[Ca74] - CABRIC, M.
Définition et traduction de langages de haut niveau par macros syntaxiques
Rapport de D.E.A., Informatique, U.S.M. Grenoble
Septembre 1974

[Cd71] - CODD, E.F.
A data base sublanguage founded on the relational calculus
IBM Research, San Jose
July 1971, No. RJ893

[Cd74] - CODD, E.F.
Recent investigations in relational data base systems
IFIP Congress
1974, pp.1017-1021 ; North Holland Pub.

[Ch66] - CHEATHAM, T.E. Jr.
The introduction of definitional facilities into higher level programming languages
AFIPS Conference Proceedings
1966 FJCC, Vol.29, 2nd. edition, pp.623-637

[Ch68] - CHEATHAM, T.E. Jr., FISCHER, A., and JORRAND, Ph.
On the basis for ELF - An extensible language facility
AFIPS Conference Proceedings
1968 FJCC, Vol.33, Part 1, pp.937-948

[Cl72] - COLMERAUER, A., et al.
Un système de communication homme-machine en français
Université d'Aix-Marseille, U.E.R. de Luminy
Octobre 1972

[Co67] - COHEN, J.
Langages pour l'écriture de compilateurs
Thèse de Sciences Appliquées, Faculté des Sciences de Grenoble
Juin 1967

[Cr68] - CHRISTENSEN, C.
An example of the manipulation of directed graphs in the AMBIT/G programming language
Interactive systems for experimental applied mathematics
(Klerer and Reinfelds Ed.)
1968, Academic Press, New-York

[Cs73] - COUSOT, P.
Une implantation de langages réductibles par macros syntaxiques
Séminaire de programmation, U.S.M. Grenoble
1973

[Cu71] - CURRIE, I.F., BOND, S.G., MORISON, J.D.
ALGOL 68-R
Working Conf. on Algol 68 implementation, Munich
July 1970, North Holland Pub. (Peck, J.E.L., Ed.)

[Cw72] - CHATELIN, Ph., et WILLIS, B.
Le mécanisme des macros syntaxiques
Rapport scientifique - Contrat CRI 70-107
Décembre 1972

[Da72] - DAHL, O.J., DIJKSTRA, E.W., and HOARE, C.A.R.
Structured Programming
Academic Press, New-York
1972

[Di68] - DIJKSTRA, E.W.
A constructive approach to the problem of program correctness
BIT
1968, Vol.8, pp.174-186

[Di69] - DIJKSTRA, E.W.
Notes on structured programming
Technical University, Eindhoven
1969, No. EWD 249

[Di72] - DIJKSTRA, E.W.
On the necessity of correctness proofs
Advanced course on Computer Systems Architecture, Alpe-d'Huez
December 1972

[Ea73] - EARLEY, J.
Relational level data structures for programming languages
Computer Science Dept, U. of California, Berkeley
March 1973

[Ea74] - EARLEY, J.
High level operations in automatic programming
Symposium on very high level languages
March 1974, SIGPLAN Notices, vol.9, No.4, pp.34-42

[Fe69] - FELDMAN, J.A., and ROVNER, P.D.
An Algol-based associative language
Communications of the ACM
August 1969, vol.12, No.8, pp.439-449

[Fe72] - FELDMAN, J.A., et al.
Recent investigations in SAIL--An Algol-based languages for Artificial Intelligence
AFIPS Conference proceedings
1972 FJCC, Vol.41

[Fi68] - FISCHER, A., and JORRAND, Ph.
BASEL, The base language for an extensible language facility
Massachusetts Computer Associates Report
June 1968, No. CA-6806-2811
also in :
International Seminar on Advanced Programming Systems, Jerusalem
July 1968, Vol.II, pp.151-200

[Fi73] - FISCHER, A.E., and FISCHER, M.J.
Mode modules as representations of domains
Proc. ACM Symposium on Principle of Programming Languages, Boston
1973, pp.139-143

[Fl68] - FLECK, M., and NEUHOLD, E.J.
Formal definition of the PL/1 compile time facilities
IBM Laboratory Vienna
June 1968, No. TR 25.080

[Gr68] - GRIFFITHS, M., and PELTIER, M.
Grammar transformation as an aid to compiler production
Centre Scientifique IBM, Grenoble
May 1968, No. FF2.0057.0

[Gr73] - GRIFFITHS, M.
Relationship between definition and implementation
Advanced course in software engineering
1973, Springer Verlay, Lecture Notes in Economics and Maths Systems

[Gr74] - GRIFFITHS, M.
LL(1) grammars and analysers
Advanced course on compiler construction, Munich
March 1974

[Ha69] - HAMMER, M.M., and JORRAND, Ph.
The formal definition of BASEL. Part 1 : Introduction
Massachusetts Computer Associates Report
August 1969, No. CA-6908-1511

[He69] - HENDERSON, D.A. Jr.
A description and definition of simple AMBIT/G - a graphical programming language
Massachusetts Computer Associates Report
April 1969, No. CA-6904-2811

[Ho66] - HOARE, C.A.R.
Record Handling
Nato Summer school, Villard-de-Lans
September 1966

[Hl71] - HOLLOWAY, G.H.
Interpreter/Compiler integration in ECL
International Symposium on extensible languages
December 1971, SIGPLAN Notices, Vol.6, No.12, pp.129-134

[Jo67a] - JORRAND, Ph.
Un langage d'édition
Séminaires de l'IMAG
Juin 1967, pp.221-276

[Jo67b] - JORRAND, Ph.
Intersection de deux langages context-free. Application à la grammaire du langage Algol X.
Rapport IMAG
Octobre 1967

[Jo68] - JORRAND, Ph.
A grammar for BASEL, an example of the use of the Interactive Syntax Processor
Massachusetts Computer Associates Report
November 1968, No. CA-6811-2213

[Jo69a] - JORRAND, Ph.
Tutorial on Algol 68
Third Annual Princeton Conference on Information Sciences and Systems
March 1969, pp.403-407

[Jo69b] - JORRAND, Ph.
Some aspects of BASEL, the BASE language for an extensible language facility
Extensible Languages Symposium, Boston
May 1969, SIGPLAN Notices, Vol.4, No.8, pp.14-17

[Jo69c] - JORRAND, Ph.
The formal definition of BASEL. Part 2 : Compiler
Massachusetts Computer Associates Report
August 1969, No. CA-6908-1512

[Jo69d] - JORRAND, Ph.
The formal definition of BASEL. Part 3 : Interpreter
Massachusetts Computer Associates Report
August 1969, No. CA-6908-1513

[Jo70a] - JORRAND, Ph., and HAMMER, M.M.
BASEL language programming manual
Massachusetts Computer Associates Report
May 1970, No. CA-7005-2111

[Jo70b] - JORRAND, Ph. and WEGNER, P.
Some aspects of the structure of BASEL
Brown University Report
January 1970

[Jo70c] - JORRAND, Ph.
Remarques sur les langages extensibles
Congrès de l'AFCET, Paris
Septembre 1970, Brochure No.3, pp.61-78

[Jo71] - JORRAND, Ph.
Data types and extensible languages
International Symposium on extensible languages
September 1971, SIGPLAN Notices, Vol.6, No.12, pp.75-83

[Jo72] - JORRAND, Ph. and BERT, D.
On some basic concepts for extensible programming languages
International Computing Symposium, Venice
April 1972, pp. 2-16

[Jo72a] - JORRAND, Ph., et SCHUMAN, S.A.
Définition d'un langage ou définition d'une implémentation ?
Congrès de l'AFCET, Grenoble
Novembre 1972, pp. 369-389

[Jo73] - JORRAND, Ph.
L'extensibilité en Algol 68
Journée Algol 68 de l'Institut de Programmation, Paris
Avril 1973, pp. IV.1-IV.13

[Jo74] - JORRAND, Ph.
"Classes" for programming objects - Potential applications
IFIP WG2.1. Working Paper
July 1974, No. B10/274

[Ko69] - KOSTER, C.H.A.
On infinite modes
Algol Bulletin No. 30
February 1969, pp. 86-89

[La66] - LANDIN, P.J.
The next 700 programming languages
Communications of the ACM
March 1966, Vol.9, No. 3, pp. 157-166

[Le66] - LEAVENWORTH, B.
Syntax macros and extended translation
Communications of the ACM
November 1966, Vol.9, No. 11, pp. 790-793

[Le74] - LEAVENWORTH, B. (Ed.)
Proceedings of a symposium very high level languages
SIGPLAN Notices
March 1974, Vol.9, No. 4

[Li74] - LISKOV, B., and ZILLES, S.
Programming with abstract data types
Symposium on very high level languages, Santa Monica
April 1974, SIGPLAN Notices, Vol.9, No.4, pp. 50-59

[Lr68] - LAUER, P.
Formal definition of ALGOL 60
IBM Laboratory Vienna
December 1968, No. TR25.088

[Lu68] - LUCAS, P., LAUER, P., and STIGLEITNER, H.
Method and notation for the formal definition of programming languages
IBM Laboratory Vienna
June 1968, No. TR25.087

[Lu69] - LUCAS, P., and WALK, K.
On the formal description of PL/1
Annual Review in Automatic Programming, Pergamon Press
1970, Vol.6, Part 3, pp. 105-181

[Mc62] - McCARTHY, J.
LISP I.5 programmer's manual
The MIT press
1962

[Mc62a] - McCARTHY, J.
Towards a mathematical science of computation
Information Processing 1962
1963, North-Holland Pub., pp. 21-28

[Mo74] - MORRIS, J.H.
Towards more flexible type systems
Colloque sur la programmation, Paris
Avril 1974, pp. 275-302

[Na60] - NAUR, P. (Ed.)
Report on the algorithmic language Algol 60
Communications of the ACM
May 1960, Vol.3, no.5, pp. 299-314

[Ne71] - NEUHOLD, E.J.
The formal description of programming languages
IBM Systems Journal
1971, No.2, pp. 86-112

[Pr71] - PRENNER, C.J.
The control structure facilities of ECL
International Symposium on extensible languages
December 1971, SIGPLAN Notices, Vol.6, No.12, pp. 104-112

[Re69] - REYNOLDS, J.C.
A set-theoretic approach to the concept of type
Working paper, NATO Conf. on techniques in software engineering,
Rome, October 1969

[Re74] - REYNOLDS, J.C.
Towards a theory of type structure
Colloque sur la programmation, Paris
Avril 1974, pp. 303-320

[Ro69] - ROVNER, P., and HENDERSON, D.A. Jr.
On the implementation of AMBIT/G - A graphical programming language
International Joint Conf. on Artificial Intelligence, Washington
May 1969

[Sc70] - SCHUMAN, S.A., and JORRAND, Ph.
Definition mechanisms in extensible programming language
AFIPS Conference Proceedings
1970 FJCC, Vol.37, pp.9-20

[Sc71] - SCHUMAN, S.A.
An extensible interpreter
International Symposium on Extensible Languages
December 1971, SIGPLAN Notices, Vol.6, No.12, pp. 120-128

[Sc71a] - SCHUMAN, S.A. (Ed.)
Proceedings of the International Symposium on Extensible Languages
December 1971, SIGPLAN Notices, Vol.6, No. 12

[Sc74] - SCHUMAN, S.A., and JORRAND, Ph.
Toward a definitional structure for an extensible language
(Submitted for publication)
August 1974

[Sc74a] - SCHUMAN, S.A.
Toward modular programming in high-level languages
Algol Bulletin
July 1974, No. 37, pp. 12-23

[Sc74b] - SCHUMAN, S.A.
Remarques non publiées

[Sr66] - STRACHEY, C.
Towards a formal semantics
Formal Language Description Languages for Computer Programming
Proc. IFIP Working Conf. on Formal Language Description
(T.B. Steel ed.) - 1966

[St67] - STANDISH, T.A.
A data definition facility for programming languages
Doctoral dissertation, Carnegie Mellon University
1967

[St69] - STANDISH, T.A.
Some features of PPL, a polymorphic programming language
Extensible Languages Symposium, Boston
May 1969, SIGPLAN Notices, Vol.4, No. 8, pp. 20-26

[ST71] - STANDISH, T.A.
PPL, an extensible language that failed
International Symposium on Extensible Languages
December 1971, SIGPLAN Notices, Vol.6, No.12, pp.144-145

[Sz73] - SCHWARTZ, J.
On programming, an interim report on the SETL project
Computer Science Dept., Courant Institute, New-York
1973

[Vi74] - VIDART, J.
Extensions syntaxiques dans un contexte LL(1)
Thèse de troisième cycle, U.S.M. Grenoble
Septembre 1974

[vW66] - van WIJNGAARDEN, A., and MAILLOUX, B.J.
A draft proposal for the algorithmic language Algol X
WG 2.1. working paper
October 1966

[vW67a] - van WIJNGAARDEN, A., MAILLOUX, B.J., and PECK, J.E.L.
A draft proposal for the algorithmic language Algol 67
Mathematisch Centrum, Amsterdam
May 1967, No. MR88

[vW67b] - van WIJNGAARDEN, A., MAILLOUX, B.J., and PECK, J.E.L.
A draft proposal for the algorithmic language Algol 68
Mathematisch Centrum, Amsterdam
November 1967, No. MR92

[vW68a] - van WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., and KOSTER, C.H.A.
Draft report on the algorithmic language Algol 68
Mathematisch Centrum, Amsterdam
January 1968, No. MR93

[vW68b] - van WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., and KOSTER, C.H.A.
Working document on the algorithmic language Algol 68
Mathematisch Centrum, Amsterdam
July 1968, No. MR95

[vW69] - van WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., and KOSTER, C.H.A.
Report on the algorithmic language Algol 68
Mathematisch Centrum, Amsterdam
February 1969, No. MR101
also in :
Numerische Mathematik
1969, Vol.14, pp. 79-218

[vW74] - van WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A.
SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., and FISKER, R.G.
Revised report on the algorithmic language Algol 68
University of Alberta, Dept. of Computer Science
March 1974, No. TR74-3

[Wa68] - WALK, K. et al.
Abstract syntax and interpretation of PL/1
IBM Laboratory Vienna
June 1968, No. TR25.082

[We70] - WEGBREIT, B.
Studies in extensible languages
Harvard University, Cambridge
May 1970, No. ESD-TR-70-297

[We71a] - WEGBREIT, B.
The ECL programming system
AFIPS Conference Proceedings
1971 FJCC, Vol. 39, pp. 253-262

[We71b] - WEGBREIT, B.
An overview of the ECL programming system
International symposium an extensible languages
December 1971, SIGPLAN Notices, Vol.6, No.12,pp.26-28

[We74] - WEGBREIT, B.
The treatment of data types in EL1
Communications of the ACM
May 1974, Vol.17, No.5, pp. 251-264

[Wi66] - WIRTH, N., and WEBER, H.
EULER : a generalization of Algol and its formal definition
Communications of the ACM
January and February 1966, Vol.9, No.1, pp.13-23, No.2, pp.89-99

[Wi71] - WIRTH, N.
Program development by stepwise refinement
Communications of the ACM
April 1971, Vol.14, No.4, pp.221-227

[Wi73] - WIRTH, N.
The programming language PASCAL Revised Report
International Summer School on structured programming and programmed structures
1973, Munich

[Wl74] - WILLIS, B.
Définition et implantation de la sémantique des langages de programmation
Thèse d'Etat, U.S.M. Grenoble
Mars 1974

[Wn69] - WEGNER, P.
Theories of semantics
Brown University report
September 1969, No. 69-10

[Wn73] - WEGNER, P.
The Vienna Definition Language
ACM Computing surveys
March 1972, Vol.4, No.1, pp.5-63