Conception descendante de machine informatique : étude et définition du langage intermédiaire et d'une machine formelle multiprocesseurs orientée vers l'exécution du langage PASCAL

Robert Fortier

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE
DOCTEUR DE 3ème CYCLE
INFORMATIQUE

Robert FORTIER

CONCEPTION DESCENDANTE DE MACHINE INFORMATIQUE

ETUDE ET DEFINITION DU LANGAGE INTERMEDIAIRE ET D'UNE MACHINE FORMELLE MULTIPROCESSEURS ORIENTEE VERS L'EXECUTION DU LANGAGE PASCAL.

Soutenue le 10 octobre 1975 devant la Commission d'Examen :

Président : J. KUNTZMANN

Jury : F. ANCEAU
C. GIRAULT
M. GRIFFITHS
G. TASSARD

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

M. Michel SOUTIF — Présidents — M. Louis NEEL

M. Gabriel CAU — Vice-Présidents — MM. Lucien BONNETAIN, Jean BENOIT

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

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Clinique de Pédiatrie et Puériculture
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BEZES Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Clinique Oto-Rhino-Laryngologique
	CHATEAU Robert	Thérapeutique (Neurologie)
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBERMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée

MM.	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DRUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de dermatologie et syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques pures
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Mathématiques appliquées
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique Générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	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	Seméiologie médicale
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	MULLER Jean-Michel	Thérapeutique (néphrologie)
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	CHEEKE John	Thermodynamique
	COPPENS Philip	Physique
	CORCOS Gilles	Mécanique
	CRABBE Pierre	CERMO
	GILLESPIE John	I.S.N.
	ROCKAFELLAR Ralph	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

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

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

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

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.	COLE Antony	Sciences nucléaires
	FORELL César	Mécanique
	MOORSANI Kishin	Physique

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

MM.	BOST Michel	Pédiatrie
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	FAURE Gilbert	Urologie
	MALLION Jean-Michel	Médecine du travail
	ROCHAT Jacques	Hygiène et hydrologie

Fait à Saint Martin d'Hères, OCTOBRE 1974.

"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 KUNTZMANN, professeur à l'E.N.S.I.M.A.G., qui a bien voulu me faire l'honneur de présider le jury de cette thèse,

Monsieur ANCEAU, maître de conférences à l'E.N.S.I.M.A.G. qui est à l'origine de ce projet et m'a aidé de ses nombreux conseils et suggestions, en particulier lors de la première année,

Messieurs GIRAULT, professeur à l'Institut de Programmation de Paris, et GRIFFITHS, maître de conférences à l'U.S.M.G. qui ont accepté de juger ce travail,

Messieurs TASSARD et HENNERON, ingénieurs de recherche à l'I.R.E.P., qui m'ont apporté une aide précieuse pour la compréhension du langage et du compilateur PASCAL,

Messieurs SCHOELLKOPF et MARTIN, et aussi toute l'équipe d'Architecture des Calculateurs de l'E.N.S.I.M.A.G. pour leur coopération et leur aide morale,

Monsieur DAABECK de l'Université d'Aarhus (Danemark) pour le lot représentatif de programmes PASCAL qui m'a permis d'effectuer mes mesures,

Et enfin Madame DIAZ qui a assuré la dactylographie de ce texte avec une grande compétence et beaucoup de rapidité, Monsieur IGLESIAS et le personnel du service reproduction du C.I.C.G. qui ont assuré la réalisation matérielle de cette thèse avec le soin habituel.

R. FORTIER

TABLE DES FIGURES

TABLE DES MATIERES

INTRODUCTION

L'évolution du matériel (LSI en particulier) et de la microprogrammation autorise actuellement une approche descendante pour la définition de nouvelles architectures de machine.

Un projet de D.E.A. pour trois personnes : J.P. MARTIN, J.P. SCHOELLKOPF et moi-même ([FMS-73]), visant à définir une machine PASCAL, fut lancé en Octobre 1972.

Le premier semestre de cette année scolaire fut consacré à l'étude du langage PASCAL et à une étude bibliographique de machines orientées vers l'exécution de langages de haut niveau.

La compréhension complète du langage fut grandement facilitée par la présence de l'équipe de l'IREP qui implantait un compilateur PASCAL sur l'IBM 360/67 du C.I.C.G.

Une version de langage intermédiaire pour PASCAL fut déterminée au cours du second semestre, ainsi que la partie traitement des déclarations du traducteur et le suiveur syntaxique. La seule mesure statique effectuée fut celle du compilateur PASCAL en PASCAL, l'unique programme disponible à cette époque.

Bien que le travail fut un travail d'équipe pour ce projet, je m'étais plus particulièrement occupé du traitement des déclarations : descripteurs, format de l'information et traducteur.

La possibilité d'évaluation sur file avec plusieurs processeurs fut envisagée a priori à la fin de cette année.

Le projet fut poursuivi l'année suivante par un contrat SESORI orienté plus nettement vers l'aspect méthodologie de conception descendante de machine ([AF-74]).

La suite de la démarche descendante, dans le cas général fut étudiée par J.P. SCHOELLKOPF, la machine PASCAL restant de mon seul domaine.

L'expérience acquise au cours de la première année servit principalement à mieux comprendre les réalisations existantes et à dégager les concepts présidant à l'élaboration de la nouvelle version du langage intermédiaire.

Enfin le lot de programmes venant de l'université d'Aarhus (Danemark) a permis de faire des mesures significatives.

Nous avons, de plus, deux remarques à formuler :

- Les remarques du premier chapitre concernant les machines orientées vers l'exécution de langage de type ALGOL, ont été effectuées a postériori.

- D'autre part, l'extension de la démarche descendante à de tels langages est donnée uniquement à titre indicatif au chapitre huit.

I.- CONCEPTION DESCENDANTE DE MACHINE INFORMATIQUE

I.- 1. GENERALITES

I.1.a) But

Cette étude est la première partie d'un projet plus vaste qui vise à dégager une méthodologie de conception descendante de machine.

Un langage est une manière relativement précise de décrire les fonctions désirées pour une future machine, il sera donc le point de départ de la méthodologie.

Le cadre le plus général d'une telle démarche consiste à partir des besoins de l'utilisateur. Dans ce cas, la formalisation de ces besoins en termes d'un langage, nous ramène au cas des machines adaptées à l'exécution d'un langage. Nous pouvons citer à titre d'exemple le début d'une telle étude: la définition d'une machine temps réel [DO-74].

La mise au point de la méthode est faite par:

. l'étude de réalisations existantes,

. la réalisation d'une machine orientée vers l'interprétation d'un langage particulier.

Cette réalisation particulière nous permettant de mettre en évidence les problèmes posés et servant de preuve à la méthodologie.

I.1.b) Choix du langage

Trois possibilités de langage de haut niveau s'offraient à nous:

- langage interprétable

L'exemple type d'un tel langage est APL. Mais outre les nombreuses réalisations de machines APL [AB-70], un tel langage ne nécessite pas la définition d'un langage intermédiaire. Tout au moins, celui-ci résulte-t-il d'une élimination de la syntaxe et d'un simple codage.

Or cette définition est une étape très importante de la méthode descendante.

La démonstration aurait donc été incomplète.

- langage créé dans le but de réaliser une machine langage.

Un exemple de cette catégorie est SYMBOL [MS-63]. Dans ce cas, le langage lui-même est influencé par son utilisation future. La démonstration aurait été faussée.

- langage de haut niveau classique.

Nous trouvons ici les langages de type ALGOL, dont les plus récents: ALGOL 68, PASCAL, PL/1.

Plusieurs machines spécialisées dans l'interprétation de langages de ce type ont été conçues, notamment l'exemple concret des machines B 5500 à B 6700 pour ALGOL 60 [BU]. Nous n'avons pas ici les inconvénients des deux premières catégories.

Nous avons donc choisi comme exemple un langage de type ALGOL parmi les plus récents, en l'occurence PASCAL. Ce choix écartant ALGOL 68 et PL/1 intervient pour plusieurs raisons conceptuelles et techniques:

. PASCAL possède des structures de données très évoluées, en particulier les "records" conditionnels. La notion de type de PASCAL est comparable à celle de "mode" d'ALGOL 68 (extensibilité).

. PASCAL utilise très largement la notation symbolique (scalaires, ensembles), ce qui le rend relativement indépendant de la machine. Ceci est important puisque la machine future sera peu influencée par les machines actuelles (pas de type de données particulières tel que "byte" ou "bit").

. Utilisation de procédures et fonctions prédéclarées. C'est en même temps, permettre une pseudo-extension du peu d'opérateurs et autoriser une définition libre des procédures d'entrée-sortie.

. La syntaxe très simple de PASCAL a une importance surtout technique.

PASCAL ayant une syntaxe LL(1), le traducteur compacteur sera plus simple à écrire. Dans la mesure où le premier pas de la méthode élimine la syntaxe et où seule la sémantique nous intéresse, ceci représente un gain de temps appréciable.

De plus un outil adapté à une telle grammaire existe, "Transformateur de Grammaire" de Messieurs GRIFFITHS et PELETIER [GR-74].

. Relations avec d'autres équipes. En effet deux équipes extérieures utilisent PASCAL.

L'équipe IREP-CICG transportait PASCAL du CDC 6600 de ZURICH sur l'IBM 360/67 de GRENOBLE. Cette collaboration nous a permis d'étudier le compilateur pour mieux connaître la sémantique du langage et de le modifier pour nos propres besoins.

L'Université d'AARHUS (Danemark) utilise PASCAL et dispose donc d'un lot important et représentatif de programmes.

De plus, l'utilisation de PASCAL sous GEMAU est prévue dans le cadre des activités du CICG de Grenoble.

En résumé, PASCAL est un langage assez complet sans être trop complexe.

En fait, les langages ALGOL 68 et PL/1 sont assez proches pour qu'une grande partie des résultats obtenus sur PASCAL puissent s'y appliquer.

I.1.c) Langages intermédiaires

La première étape de la démarche est l'obtention d'un langage intermédiaire et de la machine abstraite associée. Celle-ci est l'interpréteur du langage intermédiaire, elle en définit la sémantique.

On peut distinguer trois catégories de langages intermédiaires suivant leur destination:

- Portabilité du langage source

Un tel langage intermédiaire est soumis principalement à la contrainte d'être le barycentre d'un ensemble de machines existantes [WI-74].

- Interprétation d'un ou plusieurs langages sur une machine existante.

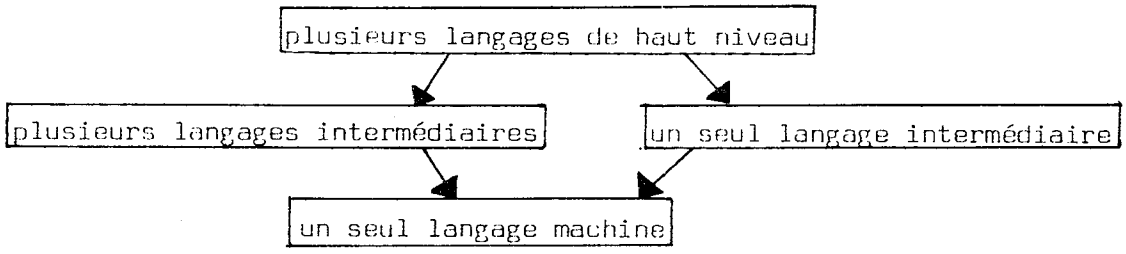
Dans ce cas la contrainte est la machine elle-même.

Un exemple est le cas de la machine B 1700, machine multi-langage où existent des interpréteurs microprogrammés des langages intermédiaires de plusieurs langages de haut niveau.

Remarque:

Depuis peu un sous-langage PASCAL est disponible sur cette machine, le langage intermédiaire correspondant est BLAISE 1726 [BE-74].

Deux cas peuvent se présenter dans cette catégorie:



- Obtention d'une nouvelle machine.

A priori il n'existe aucune contrainte inhérente à cette catégorie.

Des concepts sont donc nécessaires pour guider les choix à effectuer dans le passage du langage source au langage intermédiaire. De plus, des considérations extérieures (économiques par exemple) vont peser sur ces choix et sur la machine abstraite.

I.1.d) Notations

Au cours de la présente étude, nous donnons de nombreux exemples de correspondances entre le langage source et le langage intermédiaire.

Nous avons utilisé pour cela une notation dérivée de celle de la syntaxe produite par le transformateur de grammaire de GRIFFITHS et PETTIER [GR-74].

exemple: '<exp>' désigne l'ensemble des expressions possibles en PASCAL.

Les noms de règles sont pour la plupart identiques à ceux utilisés dans la syntaxe du langage (ANNEXE II). Avec cette notation 'if<exp> then <inst1> else <inst2>;' représente l'ensemble des instructions 'if' (avec 'else') de PASCAL.

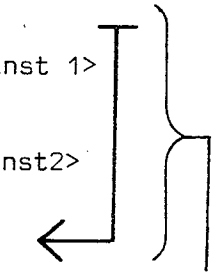
La notation du langage intermédiaire en est directement dérivée.

| <exp> désigne la traduction en langage intermédiaire de '<exp>'

Les instructions du langage intermédiaire sont en majuscules pour les différencier des terminaux du langage source.

Exemple:

PASCAL	Langage intermédiaire
if	| <exp>
<exp>	IF
then	| <inst 1>
<inst1>	FIN
else	| <inst2>
<inst2>	SORT
;	



paramètre de l'instruction IF précisant l'adresse du branchement éventuel.

I.2. LES MACHINES B 5500 A B 6700

Il ne s'agit pas ici de voir en détail le fonctionnement de ces machines, mais de retrouver les concepts qui ont guidé l'élaboration du langage intermédiaire et de la machine.

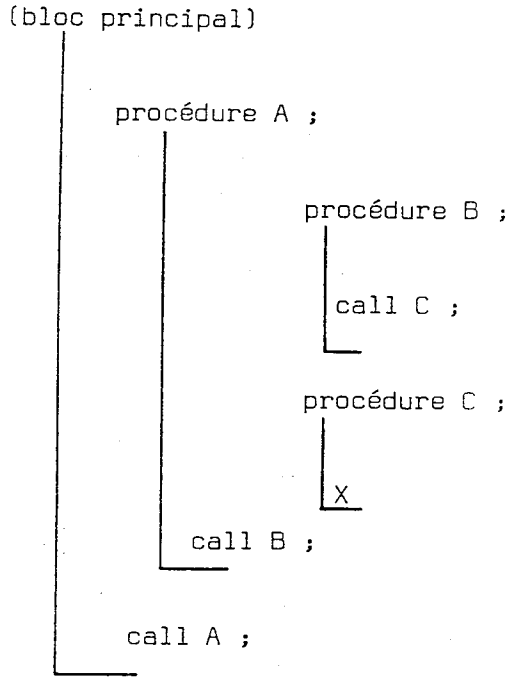
Il nous faut donc isoler d'abord la partie 'machine ALGOL' [BU], [HD-68] de la partie 'système'.

1.2.a) Structure générale de l'interpréteur

La structure récursive du langage ALGOL (Procédures), ainsi que la mise générale en post-fixé, font que ces machines sont organisées autour d'une pile double.

Une pile de contexte en mémoire contient les variables créées et manipulées et une pile d'évaluation contient les valeurs de l'expression en cours de calcul.

Le programme ALGOL suivant donne la représentation Mémoire présentée dans la figure I.1.



Les appels suivants ayant eu lieu successivement:

call A ; call B ; call C .

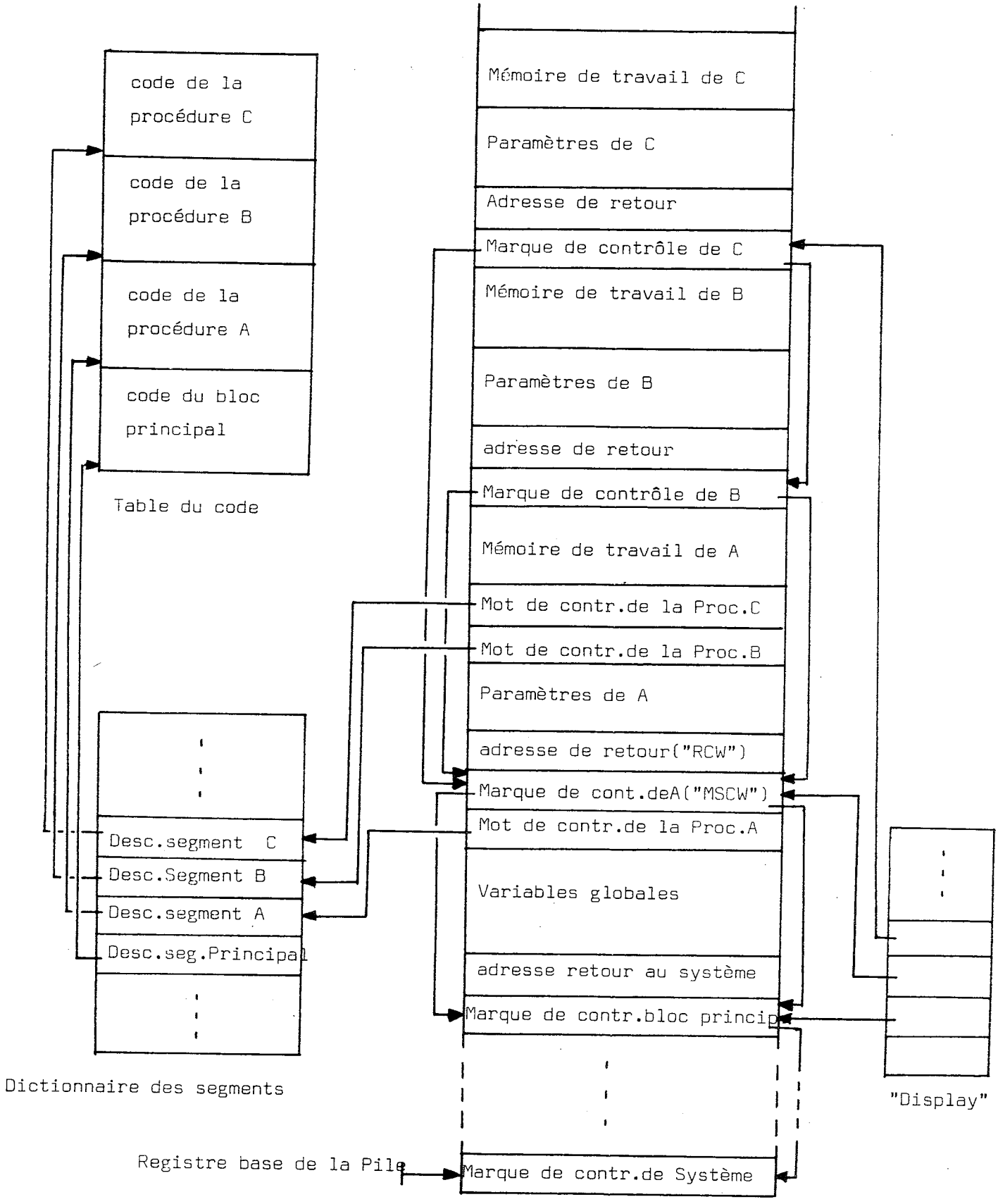


Figure I.1.. Organisation de la mémoire de la B 6700

Remarque:

Dans l'implémentation de la machine, la partie pile d'évaluation d'expression est placée au sommet de la zône de travail de la procédure en cours, à l'exclusion des sommet et sous-sommet de cette pile d'évaluation qui sont placés dans deux registres spéciaux de la machine pour accélérer l'évaluation.

L'adressage de la mémoire se fait par syllabe de 51 (48+3) bits. Un descripteur de variable ou procédure, occupe une syllabe. Toute syllabe comporte un préfixe (3 bits) indiquant la nature de son contenu Un tel mécanisme permet la renomination interne des variables par leur numéro de déclaration.

I.2.b) Déclarations

Les déclarations sont presque entièrement traitées à la traduction. Elles sont traduites par une instruction 'ENTER <p>' qui réserve globalement la place mémoire nécessaire à l'ensemble des déclarations du nouveau bloc (sauf pour les structures dynamiques (tableau) dont seul le descripteur est empilé normalement).

Les descripteurs de variable simple sont d'une syllabe et contiennent la valeur (les réels longs ont deux syllabes). Le préfixe indique le type de la variable. Les descripteurs de structures contiennent l'adresse des valeurs ainsi que des renseignements système (présence, ...).

Il existe une exception pour les chaînes de caractères. La chaîne est condensée en plaçant 6 caractères dans une syllabe. Elle s'étend sur le quotient de la division entière du nombre de caractères par six+une syllabes.

Les descripteurs de procédure (PCW) ont le même format que ceux de variable.

En ce qui concerne les valeurs, les machines B 5500 à B 6700, possèdent l'originalité de représenter les entiers comme des cas particuliers de réels (courts), ce qui est en accord avec la définition mathématique. Un entier est un réel dont l'exposant est nul. Ceci permet de n'avoir qu'un seul type d'opérateurs arithmétiques.

En conclusion, l'emploi de l'adressage par syllabe, avec un descripteur par syllabe, réduit la renomination interne (au niveau du bloc) au plus simple. La syllabe elle-même a été choisie suffisamment grande pour que le descripteur puisse contenir les valeurs des variables simples. La distinction entre entier et réel a disparu.

Il ne reste à faire à l'interpréteur que la partie dynamique des déclarations.

I.2.c) Instructions

Nous avons classé les instructions de la B6700 en plusieurs catégories suivant leur proximité avec le langage source (ALGOL 60) et les raisons de leur modification par rapport à ce même langage.

. Opérateurs arithmétiques

Ces opérateurs sont exactement en correspondance avec ALGOL (+↔ ADD ; *↔ MULT). Indépendamment du postfixage des expressions, il s'agit ici d'un simple codage.

Les instructions particulières à la double précision (MULT EXTEND, SET TO SIMPLE (DOUBLE) PRECISION) correspondent à l'emploi des réels longs dans le langage. Ce sont des anticipations sur l'affectation future.

Il y a donc là un transfert du descripteur (déclaration) à l'instruction (utilisation).

· Comparaison, affectation

Pour ces deux catégories d'instructions, on assiste à une distinction entre les cas d'une syllabe ou de plusieurs syllabes.

Il existe un jeu d'opérateurs de comparaison et une instruction d'affectation pour les variables simples (une seule syllable):

SUP, ...EQ ; STORE

D'autres opérateurs sont utilisés dans le cas des structures (plusieurs syllabes).

L'affectation dans ce cas est l'instruction: TRANSFER.

· Accès aux variables

Il existe deux versions d'instruction d'accès suivant qu'on veut accéder à la valeur ou à l'adresse de celle-ci:

LOAD	accède la valeur (variable simple),
NAME CALL	accède le descripteur.

Remarque:

Si le descripteur contient la valeur, ces instructions ont le même effet.

Dans la B 5500, ces instructions s'appellent respectivement "Operande Call" et "Descriptor Call".

L'indexation existe en trois exemplaires.

INDEX exécute l'indexation proprement dite.

Les deux autres versions résultent de contractions des couples d'instructions

INDEX, LOAD →INDEX and LOAD VALUE

INDEX, NAME CALL→INDEX and LOAD NAME

Cette contraction économise un calcul d'adresse (LOAD et NAME CALL n'aurait pas ici de paramètre).

. Affectation

L'affectation comporte en fait quatre versions de la même instruction. En plus de la taille des valeurs, il existe un autre paramètre purement fonctionnel qui est la destruction ou non du résultat qu'on vient de mémoriser.

D'où "STORE(TRANSFER) Destructive"
"STORE(TRANSFER) NON Destructive"

La non destruction est utilisée notamment dans le code produit pour une boucle FOR (affectation à la variable de contrôle de boucle).

Ceci nous amène aux instructions particulières au fonctionnement de la pile d'évaluation (non spécifiquement liées à ALGOL):
"PUSH", "EXCHANGE", "Delete Top of STACK".

. Littéraux

Dans le but de gagner de la place, il y a plusieurs versions des littéraux (vraisemblablement guidées par des mesures de fréquence statique) suivant leur taille:
LITT'0', LITT'1', LITT sur 8 bits, LITT sur 16 bits, LITT sur 48 bits.

. Opérateur pour les chaînes de caractères

Le fait de condenser ces chaînes dans les syllabes a imposé l'introduction d'une instruction spéciale: PACK (l'instruction 'UNPACK' apparait seulement comme sous-instruction d'autres instructions).

. Appel procédure

Contrairement à la plupart des instructions, l'appel de procédure est en partie préfixé. L'instruction MARK STACK provoque le changement d'environnement et empile les descripteurs marque de bloc ('MSCW' et 'RCW') avant l'évaluation des paramètres.

La séquence de langage intermédiaire pour un appel de procédure est représentée par :

Mark stack	changement d'environnement	préfixé	postfixé
| <paramlist>	évaluation des paramètres effectifs		
NAME CALL p	accès au descripteur de la procédure p, passage des paramètres et branchement.		

. Retour de procédure

S'il existe un même descripteur pour les procédure et fonction, ceci impose deux instructions de retour différentes.

EXIT retour normal de procédure

RETURN retour normal de fonction avec mise du résultat dans la pile d'évaluation.

. Branchements

Le 'GOTO' d'ALGOL est réduit à la traduction à un branchement à une adresse littérale dans le code.

. Instructions de contrôle (if, for, while...)

Ces instructions sont réduites après la traduction à leur plus simple expression, qui est un branchement conditionnel et un branchement inconditionnel.

Exemples:

ALGOL	Langage intermédiaire
if	| <exp>
<exp>	LITT 'n' n: déplacement dans le code de l'alternative fausse
then	
<inst>	BRANCH FALSE branchement si faux à:adresse courante+'n'
;	| <inst>

while	| <exp>	
<exp>	LITT 'n'	n déplacement pour sortir de la boucle
do	BRANCH FALSE	
<inst>	| <inst>	
	LITT m	m déplacement pour revenir au début de la boucle
;	BRANCH	branchement inconditionnel

En résumé, une instruction du langage intermédiaire peut avoir plusieurs provenances:

1/ C'est une partie commune à plusieurs instructions du langage source (exemple: BRANCH FALSE pour if, while...)

2/ Elle vient directement du langage source (ex.: ADD).

3/ C'est un cas particulier d'utilisation d'une instruction du langage source, soit en raison de la fréquence de ce cas (ex.: LITT'0'), soit en raison de l'implantation (ex.:TRANSFER).

4/ Elle appartient à un ensemble de plusieurs instructions représentant une instruction du langage source (ex.: MST).

5/ Elle est introduite par le mécanisme d'interprétation choisi (ex.: PUSH): remontée de primitives fonctionnelles.

Nous voyons donc que ces instructions proviennent d'un traitement complexe du langage source (sauf dans le cas 2/) en plusieurs phases.

Deux paramètres déterminent ces phases:

- la fréquence,
- la machine abstraite (remontée des primitives).

Nous pouvons aussi remarquer qu'ici les déclarations sont traitées à la traduction et qu'une partie du descripteur des variables est inclue dans le code.

I.3. MACHINE PL

L'intérêt de cette machine réside dans l'utilisation du concept de segment et l'emploi très large des mesures statiques et dynamiques [WO-72].

Le langage PL est un sous-ensemble de PL/1 qui ne conserve que les notions classiques communes avec ALGOL, avec de plus, les variables "Controlled" et les chaînes "Varying". Il n'y a ni 'structure', ni variable 'Based' ni fichier.

I.3.a) Structure de l'interpréteur-notion de segment

Comme pour les machines BURROUGHS, la machine PL est organisée autour de deux piles regroupées, pour les mêmes raisons.

La différence essentielle est le découpage du programme en ce que D.B. WORTMAN appelle 'Segment'.

Un segment est un ensemble d'instructions du langage intermédiaire qui correspond à une entité du langage source. Le corps d'une procédure est un segment, une alternative d'un "if", le corps d'un groupe 'do' en sont aussi.

Grossièrement, les mots clés du langage source déterminent les bornes de ces segments (Begin, end, do, then, else...).

Le séquencement et le contrôle d'un programme se font par l'intermédiaire des descripteurs de segments et non par des littéraux 'adresse relative'. Sauf, bien entendu, pour le 'goto' qui est un branchement à une adresse littérale du programme ou à une adresse contenue dans une variable étiquette.

Le nom du segment est son numéro d'ordre, et le descripteur de segment contient l'adresse de la première instruction de ce segment.

Les déclarations de PL sont traduites en une table de descripteurs de variables du préfixe 'indéfini' et type de la variable (Float, Fixed, Char...): (table des symboles).

Lors de l'entrée dans un nouveau bloc, une instruction particulière transfère l'ensemble des descripteurs correspondants de la table des symboles dans la pile de contexte.

Les descripteurs de variables sont ici plus élaborés que dans les machines BURROUGHS, le préfixe contenant le type précis de la variable.

Remarque:
Le bloc est la procédure de PL/1.

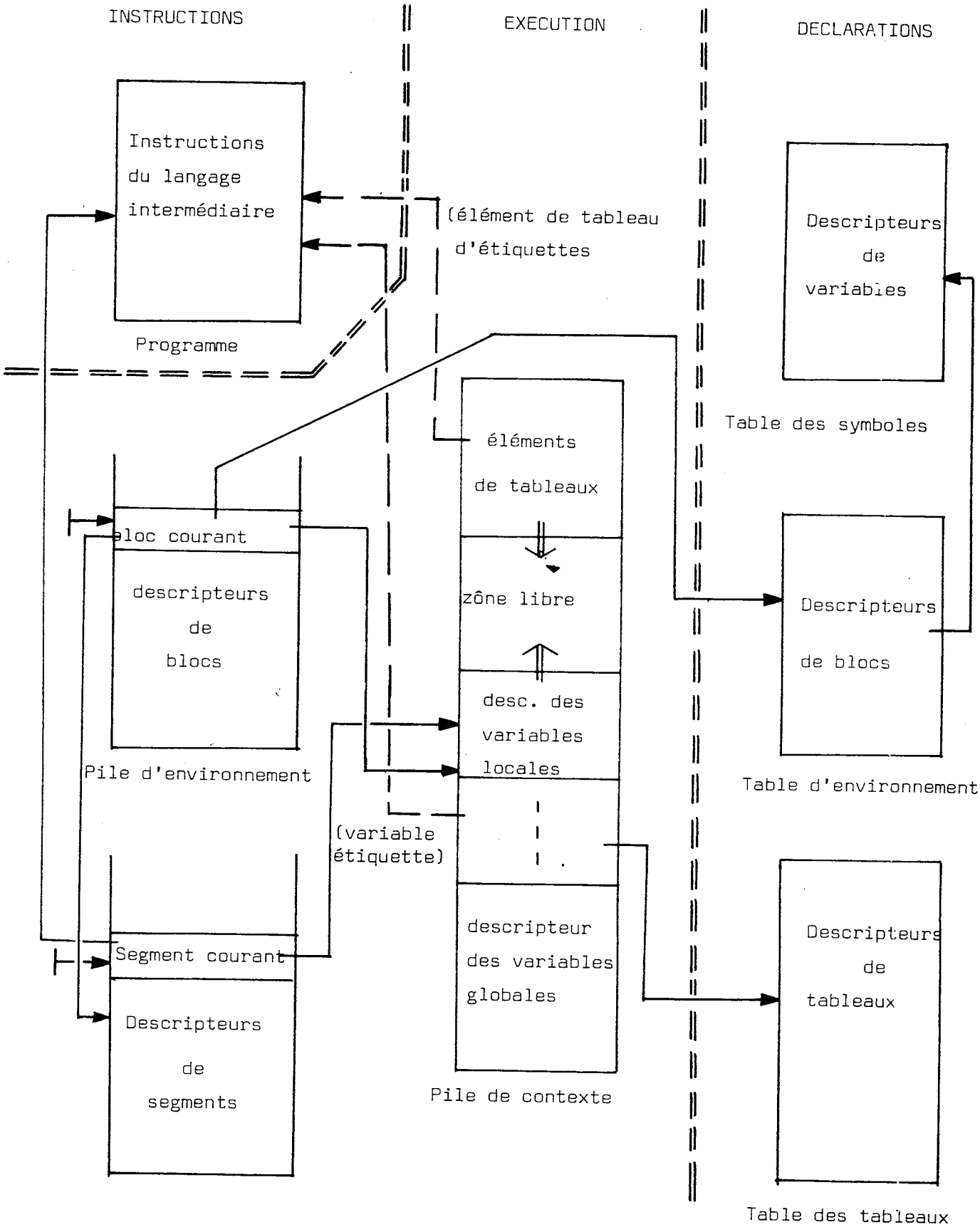


Figure I.2. Organisation de la Mémoire de la machine PL

I.3.b) Instructions

Les instructions du langage intermédiaire ont été déterminées en plusieurs étapes.

Un premier jeu de primitives a été choisi à partir du langage source et compte tenu de la notion de segment.

Les différences essentielles avec l'exemple précédent tiennent en trois points:

- les descripteurs de variables contiennent une information de type plus complète, d'où une seule version des accès, des affectations et des descripteurs.

- la segmentation permet aux instructions de contrôle et de manipulation des descripteurs de segment d'être plus proches du langage source.

- la table des symboles entraîne l'existence d'instructions déclaratives pour les blocs (SCOPEID) et non plus une simple réservation de place.

De plus, dans cette version, il y a très peu d'instructions fonctionnelles: 'POP' et 'SWAP' pour la pile d'évaluation et 'BIT' pour la conversion en bits (Push et Pull apparaissent dans les instructions 'EVAL' et les opérateurs).

Les instructions de PL pour les variables "controlled" (Allocate et Free) sont traduites par ALLOC et FREE.

Pour les paramètres, les déclarations 'ENTRY' de PL introduisent une instruction 'PARAM' qui contrôle le type des paramètres effectifs et force éventuellement une conversion (Argument muet).

En ce qui concerne l'accès, il existe une instruction accédant au descripteur (si celui-ci contient la valeur et non son adresse, elle est

remplacée par cette adresse) 'NAME'.

Dans les expressions où la valeur est nécessaire, une deuxième instruction (EVAL) accède à celle-ci par l'intermédiaire du descripteur.

Le nom interne des variables est le couple: niveau statique (S), numéro de déclaration (D). L'accès à la valeur de la variable A est alors effectué par 'NAME S_A,D_A' suivi de 'EVAL'.

Exemple d'instruction de contrôle:

| <exp>
|---|---|
| BIT | conversion en bit (0/false, 1:true) |
| NAME <ad.seg.alternatives> | empile les descripteurs des 2 alternatives |
| SWAP | échange ces deux descripteurs |
| SUBS 1 | indexe ces descripteurs avec le résultat du test |
| EVAL | accède l'adresse dans le descripteur sélectionné |
| CALL | branchement à l'alternative(segment) |

I.3.c) Optimisation

La méthode employée pour optimiser le langage intermédiaire consiste à mesurer statiquement et dynamiquement la fréquence de ses instructions et surtout les rapports entre elles.

On mesure cette fréquence pour les paires et les triplets d'instructions, statiquement.

Des mesures statiques par niveau lexicographique permettent de minimiser la place prise par le nom interne (optimisation du codage).

Parmi les résultats les plus marquants on trouve la très grande fréquence de la paire 'NAME', 'EVAL' et l'importance du niveau global pour les références.

Ceci amène les modifications suivantes:

. la paire 'NAME', 'EVAL' est remplacée par une instruction 'LOAD',

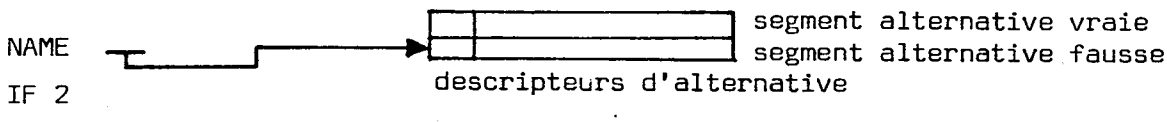
. la fréquence par niveau lexicographique introduit la notion de nom court et nom long.

La paire 'STORE', 'POP' est pour les mêmes raisons, transformée en l'instruction STORED.

D'où les deux instructions STORE (Store non Destructive, Transfer non Destructive de la B 6700) et STORED (Store destructive de la B 6700).

Dans le cas des instructions de contrôle, les regroupements sont plus importants et s'accompagnent d'une légère diminution de l'imbrication des segments (2 segments imbriqués pour 'do while' au lieu de 3).

La traduction de l'instruction 'if' devient alors:



Remarque:
Il existe une instruction 'IF 1' qui correspond au 'if' sans 'else'. Cette séparation est due à la grande fréquence dynamique de ces instructions (les plus fréquentes parmi les instructions de contrôle).

Le reste des instructions est affecté dans une moindre mesure par ces résultats.

On obtient alors le langage intermédiaire définitif.

Des mesures comparées entre ces deux machines successives permettent alors de chiffrer les gains en place et en temps.

Exemple:

La deuxième version de langage intermédiaire apporte par rapport à la première, pour la traduction de l'instruction 'if' seulement:

- 1,9% de réduction d'espace mémoire,
- 9,8% de réduction de référence mémoire pour les instructio
- 8,6% de réduction du nombre de bits accédés pendant l'exéc tion.

ceci en moyenne sur les programmes mesurés avec les deux versions.

Le gain total d'espace mémoire obtenu en utilisant la deuxième version du langage intermédiaire au lieu de la première, est de 51,1%, ce qui est assez considérable

I.4. METHODE DE GREBERT

I.4.a) Généralités

Il s'agit de la méthode mise au point par GREBERT pour la machine SPLM [GR-72] et illustrée en partie par un exemple TOY.

L'idée majeure est de faire subir le minimum de transformation au langage source.

Dans ce but, le premier pas consiste à éliminer la syntaxe, coder les informations et éventuellement réordonner (post ou préfixé) les symbole de base du langage source (méthode également utilisée dans les compilateurs incrémentiels [BER-74]).

Remarque:
Les symboles de base n'ayant qu'une signification syntaxique sont élimi nés.

L'écriture d'un traducteur-compacteur permet alors des mesures statique pour optimiser le codage, puis un interpréteur du langage intermédiaire (machine abstraite) est écrit. Il permet des mesures dynamiques.

Il introduit aussi un certain nombre de variables de travail qui seront des variables de la future machine.

L'implantation de ces variables est en partie déterminée par les mesures dynamiques (Mémoire, registre, mémoire locale).

Bien entendu quelques retours arrière dans la démarche descendante sont permis, mais ils ne doivent consister qu'en des changements d'options sans réellement violer le principe de cette démarche.

I.4.b) Langage TOY

Ce langage est un langage très simple avec peu d'instructions.

Le langage intermédiaire obtenu par cette méthode est très proche du langage source. Seuls sont éliminés la syntaxe et le symbole ',' séparant les déclarations. Une mise en post-fixé générale et le codage complètent la traduction.

Le ';' transformé en l'instruction 'PVG' a ici un rôle sémantique original. C'est un opérateur de séquencement. Dans le cas simple, il exécute le passage à l'instruction suivante. Mais dans celui où un test a été fait, et a laissé dans la pile d'évaluation un descripteur d'étiquette, il provoque le branchement à cette étiquette (ANNEXE VI, Langage TOY).

Le 'DPT' (':') provoque l'affectation à la variable étiquette. Dans cet exemple, aucun contrôle sémantique n'est effectué à la traduction. Les descripteurs de variables sont donc très complets pour la détection dynamique des erreurs.

Remarque:

La démarche descendante pour TOY a été poursuivie jusqu'à la simulation d'une machine TOY en CASSANDRE et la microprogrammation sur MITRA 15 de l'interpréteur du langage TOY [AN-74].

Cette méthode amène deux questions devant être résolues dans le cadre des langages de haut niveau:

. Quels sont les critères permettant de choisir si un symbole de base est redondant?

. Comment choisir l'ordre de réordonnancement préfixé, infixé ou post-fixé, pour une instruction du langage source.

I.5. METHODOLOGIE

Pour répondre à ces deux questions, nous avons introduit un nouveau concept de choix: le traitement procédural ou opératif des instructions du langage source.

I.5.a) Méthode procédurale

Cette méthode se caractérise par l'introduction de la notion d'environnement(ou contexte).

Grossièrement, elle se traduit par une mise en préfixé et le traitement de l'instruction du langage source, en tant qu'"appel de procédure".

Dans une première approximation, le nom de l'instruction est l'appel de la"procédure", les expressions, ou groupe d'instructions, de celle-ci sont le "corps de cette procédure".

Un exemple trivial d'un tel traitement est une opération du langage source: A+B devient Plus(A,B).

D'où la forme définitive

```
PLUS        (environnement d'addition)
VARIABLE A
VARIABLE B
```

En fait, un appel de procédure regroupe deux notions séparées et complémentaires:

. gestion environnement (Préfixé): Nom de la procédure

. opération (classiquement post-fixée): Corps de la procédure.

De plus, il existe plusieurs sortes d'environnement dans un appel normal de procédure:

[contrôle (lié au code)
[adressage (lié aux noms internes)

Cette méthode s'applique principalement à une instruction entière du langage source.

Une conséquence immédiate de la création d'un environnement est la particularisation de certaines instructions du langage intermédiaire appartenant à cet environnement.

Ce concept procédural introduit une segmentation analogue à celle utilisée par WORTMAN dans PL (I.3.).

Un segment sera ici l'ensemble des instructions du langage intermédiaire soumis à l'environnement créé, c'est-à-dire le "corps de la procédure".

Les exemples suivants vont nous permettre de préciser cette méthode:

- if <exp> then <inst1> else <inst2>;

Remarque:
Il s'agit ici uniquement d'un environnement de contrôle.

La "procédure" a la forme:

IF (<exp> then <inst1> else <inst2>) ;
↑ ↑
appel corps fin de la procédure

Les opérations de cette procédure sont uniquement des opérations de séquencement.

Le point virgule (caractère semi-redondant) a dans ce cas la même signification qu'un retour de procédure, c'est-à-dire retour dans l'environnement d'"appel". L'adressage de retour est ici le passage en séquence simple.

'Then' est l'opérateur de séquencement qui a comme opérande l'expression et qui rend comme résultat l'adresse de l'instruction à exécuter.

'Else' marque la fin de la première alternative, c'est une sortie anticipée de la procédure.

La traduction donne donc:

Segment "IF"	IF	entrée dans le segment 'IF'
	|<exp>	
	THEN	opération de séquencement (test et branchement)
	|<inst1>	
	ELSE	fin de la première alternative('exit'de la procédure)
	|<inst2>	
	PVG	

Un autre exemple d'instruction du langage source, nous montre une signification un peu particulière du point virgule (signification qui existe un peu pour le test dans TOY)

<var>:= <exp>; devient AFFECT (<var><exp>) ;

D'où:

Segment 'AFFECT'	AFFECT	entrée dans l'environnement affectation
	|<var>	instructions d'accès éventuellement modifiées pour obtenir l'adresse valeur
	|<exp>	
	PVG	Mémorisation et sortie de l'environnement

L'instruction d'un langage de haut niveau qui regroupe à la fois les environnements de contrôle et d'adressage est l'appel de procédure :

<Nom Proc>(<exp1>,..., <exp$_n$>);

la transformation nous donne :

APPEL (<NomProc>,<exp1>,..., <expn>);

D'où:

segment "appel"	APPEL	environnement 'appel de procédure'
	<Nom Proc>	accès au descripteur de procédure
	<exp1>	calcul des paramètres effectifs
	<expn>	
	PVG	exécution proprement dite de l'appel et sortie de l'environnement 'appel de procédure' pour rentrer dans celui de la procédure elle-même.

Remarque:

Nous voyons qu'avec cette méthode, le point virgule n'est pas considéré comme caractère redondant.

I.5.b) Méthode opérative

Cette méthode est beaucoup plus simple. Il s'agit de considérer le nom de l'instruction comme opérateur qui agit sur les autres composants considérés comme opérandes. L'instruction est alors mise en post-fixé et les autres symboles de base, considérés non significatifs, sont éliminés.

Exemple:

A+B devient
- Variable A
- Variable B
- PLUS

Cette méthode a une portée localisée, elle s'applique à une partie ou à toute une instruction.

Les exemples précédents traités par cette méthode donnent:

Instruction 'if':

if	|<exp>	
<exp>	IF	opérateur de séquencement ayant l'expression comme opérande et retournant l'adresse sélectionnée
then	|<inst1>	
<inst1>	ELSE	branchement
else	|<inst2>	
<inst2>		
;	(PVG)	séquencement

<var>:= <exp>; devient <var><exp> AFF(PVG)

Ici se pose le problème d'obtenir l'adresse de la variable à affecter et non sa valeur. Or rien ne l'indique au moment de l'accès à cette variable. Ceci doit donc être résolu à la traduction et une instruction de calcul de l'adresse seule doit être introduite dans le langage intermédiaire ('NAME CALL' de la B6700 et 'NAME' de la machine PL).

Appel de procédure:

A	Nom A
(	
<exp1>	|<exp1>
,	⋮
⋮	
<expn>	|<expn>
)	APPEL
;	(PVG)

Remarque:
Cette méthode considère en général le point virgule comme un caractère redondant (sauf pour les instructions 'For' et 'while').

I.5.c) Comparaisons de ces méthodes

On constate que la méthode opérative donne moins d'instructions de langage intermédiaire pour une instruction du langage source, que la méthode procédurale (2 instructions contre 3 pour l'appel de procédure par exemple).

Mais la méthode procédurale avec son environnement de contrôle donne une sécurité plus grande (notamment pour les 'goto exit' de PASCAL) et peut réduire le nombre total des instructions du langage intermédiaire puisque une instruction du langage intermédiaire peut avoir plusieurs sens suivant l'environnement dans lequel elle est utilisée (ex.:PVG).

En conclusion, nous emploierons plutôt la méthode opérative si aucune détection d'erreur n'est nécessaire, et la méthode procédurale si nous voulons une sécurité plus grande.

Un compromis entre ces méthodes est bien entendu possible.

Les exemples suivants nous montrent concrétement les emplois de ces deux méthodes dans le cadre des machines existantes.

Méthode	appel de procédure	instruction WHILE
PROCEDURALE	Machine PL NAME(p) accès au desc. de Segment environnement de contrôle de procédure |<exp1> (PARAM) évaluation des paramètres . contrôle du type et création . éventuelle d'un argument muet . |<expn> (PARAM) ENTER entrée dans l'environnement d'adressage de la procédure, passage des paramètres et branchement au début du code	dans machine PL LOAD accès au desc.de segment CALL [entrée dans l'environnement de contrôle de boucle |<exp> CRET test: si faux sortie LOAD] branchement à la liste CALL] d'instructions du while CYCLE fin segment(bouclage)
OPERATIVE	dans HP 3000 (HP) |<exp1> . évaluation des paramètres . effectifs considérés comme . opérandes |<expn> PCAL p [changement d'environnement et branchement au début du code(opération n-aire)	dans B 6700 |<exp> BFC saut si le test est faux |<inst> B bouclage

Compromis entre les deux méthodes:

Appel de procédure de la B 6700

MS préparation du changement d'environnement d'adressage
(mais ce changement n'a pas lieu ici)

|<exp1>

|<expn>

NAME CALL p: accès au descripteur, passage des paramètres changement des environnements et branchement.

I.5.d) Méthodologie dégagée

A partir de la méthode de GREBERT et compte tenu de celle employée par WORTMAN et dans les machines BURROUGHS, et les concepts dégagés plus avant, nous pouvons représenter la méthode à employer pour PASCAL par le schéma suivant.

Les contraintes extérieures ont été volontairement séparées de la méthode elle-même.

L'application au langage PASCAL dans les chapitres suivants précise les détails de ce schéma général.

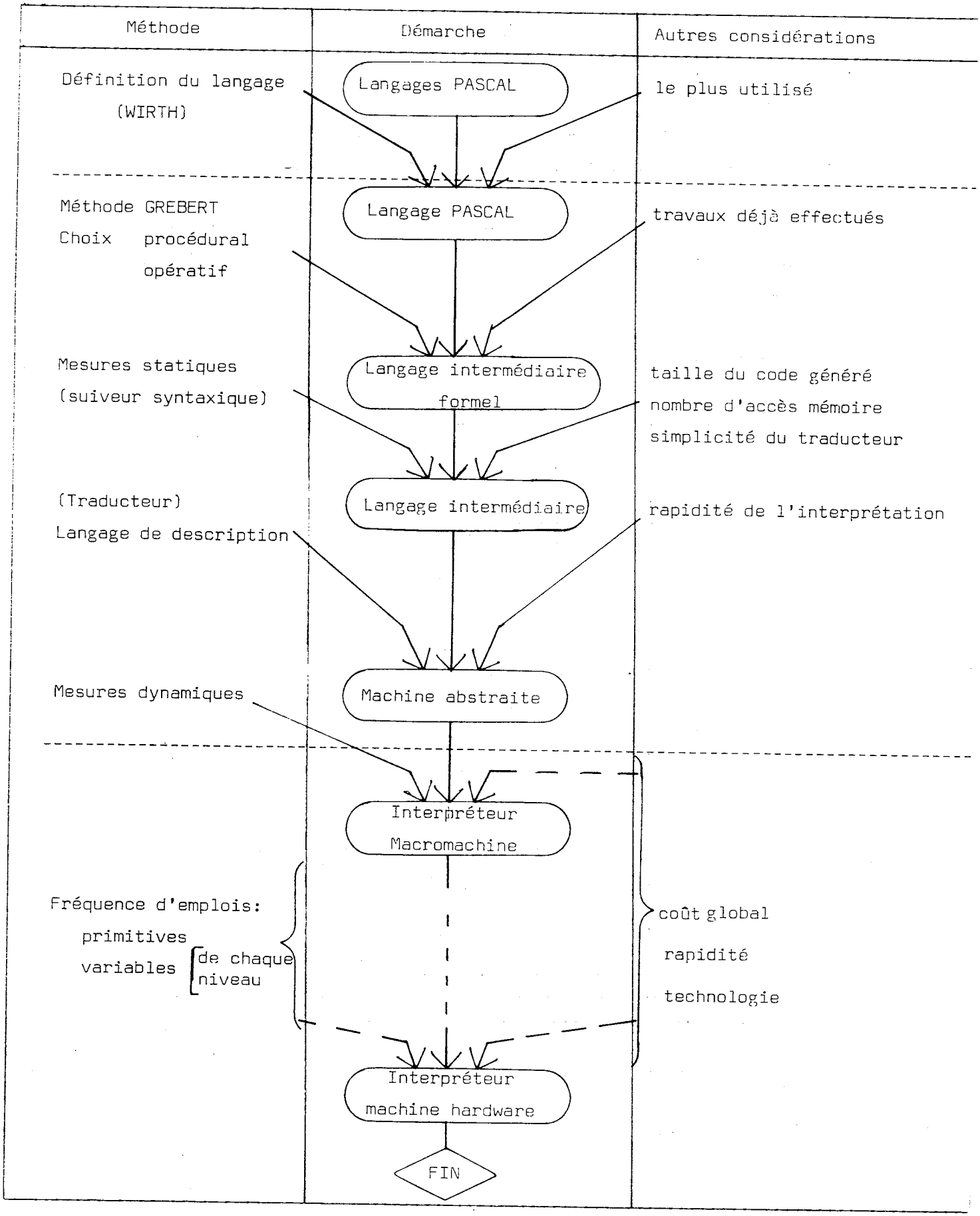


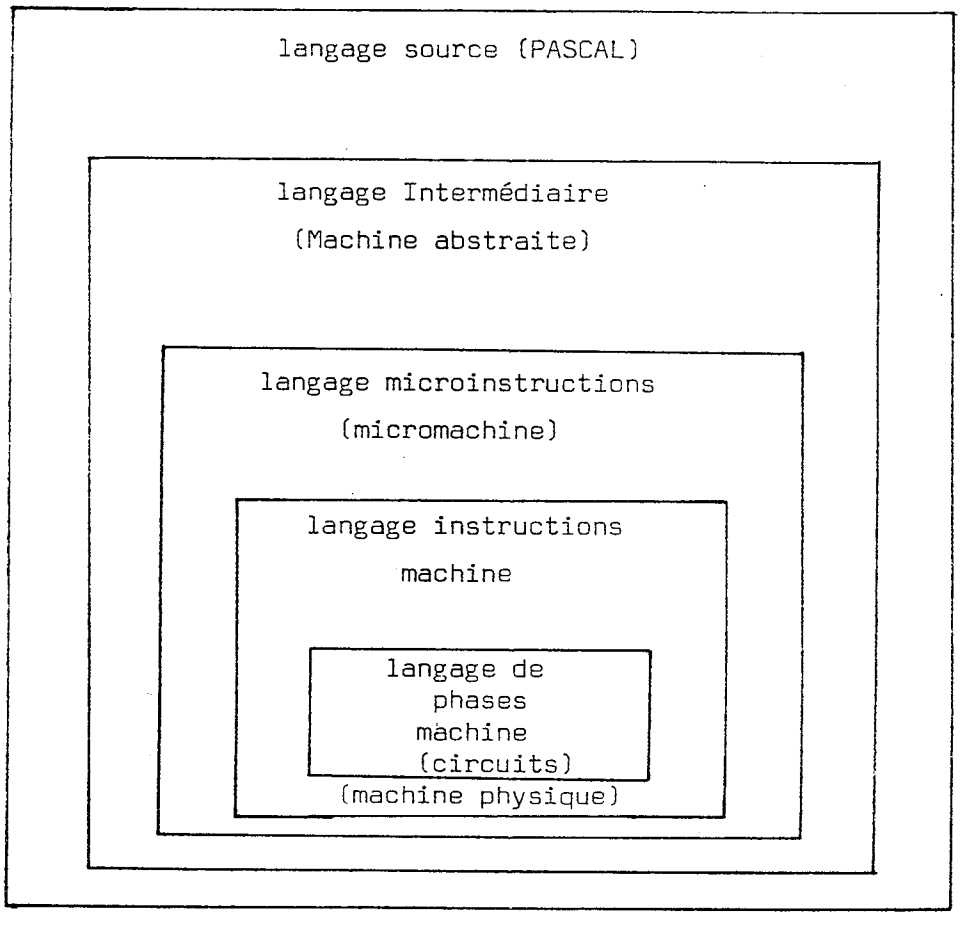
Figure I.3.: Schéma de la démarche de conception descendante appliquée à PASCAL

D'autre part, le concept descendant de la démarche introduit une notion de niveau.

Nous parlerons de niveau d'interprétation, dans la phase trois, pour désigner les interpréteurs successifs du langage intermédiaire.

Ces interpréteurs s'appliquent à des langages de plus en plus fins. Le premier étant le langage intermédiaire, le dernier le langage de phase de la machine.

Un exemple d'imbrication:



Le nombre d'étapes peut être réduit éventuellement.

Le degré d'imbrication est le niveau du langage obtenu. A un niveau donné,

les instructions du langage du niveau suivant sont appelées "primitives".

Il s'agit ici d'une terminologie.

Les instructions de ce niveau donné, étant chacune composées d'une ou de plusieurs instructions du niveau suivant (primitive), le choix des primitives est guidé par les mesures dynamiques et les contraintes "économiques" (ainsi que l'implantation des variables de travail introduites par l'écriture de l'interpréteur).

I.5.e) Conclusion

La méthode globalement décrite dans la figure I.3. est représentée par les chapitres II à VII. Dans la mesure du possible, chaque encadré de cette figure I.3. a donné lieu à un chapitre distinct de même nom.

Les chapitres concernant les mesures statiques et dynamiques ont été insérés à la place indiquée dans la figure.

La méthode employée pour la machine PASCAL est donc maintenant implicite dans la succession des chapitres suivants. Les concepts introduits dans ce paragraphe seront détaillés sur leur application au langage PASCAL.

En quelque sorte, la machine PASCAL est représentée par le fond des chapitres suivants, la démarche descendante est dans la forme.

II.- LANGAGE PASCAL - LANGAGE INTERMEDIAIRE

II.1. INTRODUCTION

Le langage PASCAL choisi est globalement celui défini par WIRTH en 1970 [WI-70] et revisé en 1972 [WI-72].

Les différences essentielles sont:
- la notion de variable de type 'classe' est conservée alors qu'elle a disparu dans la dernière version du langage [WI-74].
- les contraintes d'implémentation sur le CDC 6000 n'ont pas été retenues dans le but d'accepter les différentes implémentations du langage effectivement utilisées.

Les autres modifications consistent en des séparateurs différents et des mots clés différents ('CONSTANT' et 'VALUE' pour les paramètres formels par exemple).

Les fonctions et procédures standards aussi changent suivant les implémentations.

La suite de ce chapitre précisera les options choisies.

Nous reviendrons sur les caractéristiques du nouveau langage PASCAL au chapitre VIII [WI-74].

II.2. DECLARATION

Dans cette première phase, nous restons aussi près que possible du langage PASCAL, en accord avec la méthode GREBERT et les choix décrits au chapitre précédent.

Les déclarations seront traitées comme des instructions de créattion de descripteur. Le descripteur contient l'information significative de la déclaration (sous forme codée), les séparateurs sont éliminés.

C'est,en beaucoup plus complexe, un traitement similaire à celui appliqué au langage TOY pour les déclarations.

II.2.a) Blocs

La notion de bloc en PASCAL est confondue avec celle de procédure (ou fonction). Le programme principal étant une procédure sans tête, et les procédures, fonctions et types standards étant considérés comme déclarés dans un bloc englobant le programme.

Toutes les déclarations se font en début de bloc et ont pour portée le bloc.

La renomination interne sera donc, à priori, la même pour tous les objets déclarés. Cette nomination est classiquement une information sur le bloc (niveau statique) et une information sur la place dans le bloc (numéro de déclaration).

II.2.b) Etiquettes

- Etiquette simple: elle n'est pas déclarée et sa portée est la procédure dans laquelle elle est définie, (inconnue dans les procédures englobées et englobantes).

C'est en fait une "adresse immédiate" dans la procédure. Il n'y aura donc pas de nom interne, mais une valeur immédiate pour les branchements ('goto') à cette étiquette.

- Etiquette externe: elle est déclarée en début de bloc.

Cette déclaration se traduit par une renomination et création d'un descripteur qui contient la valeur de l'étiquette externe (adresse dans le code où elle est définie).

Remarques:

Toutes les instructions pouvant être étiquetées, il n'est pas interdit, à priori, de rentrer dans une boucle, dans un 'with' ou dans une alternative d'un 'IF'.

Dans les cas de la boucle 'For' et du 'with', le résultat de ce saut est complétement indéfini (aléatoire).

Un contrôle nous parait souhaitable. Il peut être fait facilement à la traduction pour les étiquettes simples.

Ce contrôle est un peu plus complexe pour les étiquettes externes, étant donné le mode de génération utilisé dans le compilateur. De plus ces étiquettes étant rares, il peut être laissé à l'interprétation.

Il suffit d'une information dans le descripteur de l'étiquette externe pour permettre ce contrôle.

II.2.c) Constantes: <ident> = <valeur>.

Comme tous les identificateurs, les identificateurs de constantes sont déclarés et ont pour portée le bloc.

Un descripteur de constante contiendra la valeur et une information de "lecture seule".

II.2.d) Types

Les types peuvent être déclarés explicitement dans une déclaration de variable. Nous les considérons alors comme ayant un nom vide.

Le type fichier, en raison de son aspect dépendant de la machine, est traité à part (cf. chapitre VIII), avec les 'procédures de manipulation associées. Seule reste ici la partie indépendante de la machine.

II.2.d.1) Scalaires: (<liste d'identificateurs>)

Cette liste est de plus ordonnée croissante. La relation d'ordre entre les identificateurs du type scalaire est leur ordre d'apparition (gauche à droite).

Les identificateurs composant le 'type scalaire' ne peuvent être considérés comme des déclarations, bien que leur portée soit le bloc.

Ils seront plutôt pris comme des valeurs symboliques qui seront représentés par des entiers (0 pour le premier, i pour le i+1°), au niveau interne, (codage). Dans ce cas, le descripteur de type contiendra la valeur maximum (le minimum est toujours 0) et une information pour faire éventuellement les transformations: forme interne (entier) ↔ forme externe (symbolique).

II.2.d.2) Intervalles ("subranges")

Ce type désigne un sous-type d'un type scalaire, en précisant les bornes:

<borne inf> . . <borne sup>

Le type scalaire de base est implicite: c'est celui auquel appartiennent les deux bornes.

Comme pour le type scalaire normal, les identificateurs (si ce n'est pas un intervalle d'entier) seront représentés par des entiers (codage).

Nous avons ici 3 informations à conserver dans le descripteur:

- la borne inférieure,
- la borne supérieure,
- le type de base (pour le codage-décodage et les compatibilités de types).

Remarque:

Les versions actuelles de compilateurs n'admettent que les intervalles entiers.

II.2.d.3) Scalaires prédéclarés: (entier, réel, caractère)

Les entiers et réels sont considérés en PASCAL comme des scalaires particuliers, dont les bornes sont définies par l'implémentation du langage sur une machine.

A chaque implémentation du langage correspond donc un type entier et un type réel particulier à la machine et prédéclarés par le compilateur (entier: 32 bits sur IBM 360, 60 bits sur CDC 6600).

Il en va de même pour le type caractère. C'est l'ensemble des caractères disponibles sur la machine, avec la relation d'ordre que celle-ci lui associe.

Remarque:

La relation d'ordre entre caractères diminue beaucoup la portabilité du langage.

II.2.d.4 - Ensembles ("powerset")

Ce type désigne: $\mathcal{P}_{(E)}$ si E est le type scalaire de base.

Une variable de ce type prend ses valeurs dans l'ensemble des parties de l'ensemble E.

E peut être tout type scalaire (sauf entier et réel) ou intervalle de scalaire.

L'information à conserver dans le descripteur est donc la même que pour les deux types précédents. Seul le codage des valeurs change.

II.2.d.5. - Tableau ("array")

C'est une collection d'éléments tous de même type.

array [<typindexlist>] of <typélément>

avec: <typindexlist> ::= <typindex> (,<typindex>)* (*: 0 ou plusieurs fois

Un tableau est donc caractérisé par:
- sa dimension (nombre d'index)
- le type de ses éléments,
- le type de son (ou ses) index.

Remarque:

La définition du tableau multidimensionnel est récurrente:

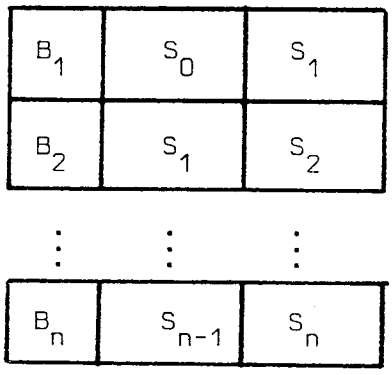
$$T_n = \text{array} \underset{n \text{ index}}{[,\ldots,]} \text{ of } t \equiv \text{array } [\,] \text{ of } T_{n-1}$$

et

$$A[I, J] \equiv A[I][J]$$

Les index appartenant au type scalaire (sauf entier et réel) ou intervalles de scalaire, les bornes du tableau sont parfaitement connues à la traduction et sont invariantes à l'interprétation.

Dans ces conditions, l'utilisation d'un pseudo-vecteur permettra de préparer les indexations en effectuant une seule fois,à la traduction, une partie des calculs.



S_n = tailles des éléments du tableau

B_i = borne inférieure de la dimension i

Si on appelle bs_i la borne supérieure de la dimension i on a:

$$S_k = (bs_{k+1} - b_{k+1}) \cdot S_{k+1}$$

L'indexation de tableau par les index I_1, ... I_n se fait alors par

l'adresse (racine) + $\sum_{k=1}^{n} (I_k - b_k) \cdot S_k$

Remarques:

S_0 est la taille totale du tableau.

Le contrôle de dépassement de borne au rang i se fait grâce à b_i et S_{i-1}

Maintenant le descripteur de tableau ne contient plus que:

- une information pour retrouver le type de ses éléments,
- une information pour retrouver le pseudo-vecteur.

II.2.d.6 Fichier

Indépendamment de l'implémentation, le seul renseignement disponible est le type des éléments (plus le facteur de blocage dans certaines versions). Il faudra y ajouter les renseignements qui ont trait à l'implémentation particulière.

II.2.d.7 Classe et pointeur

Les classes sont les seuls structures dynamiques de PASCAL. Elles sont allouées en pile.

C'est une collection d'éléments de même type. Ces éléments sont accessibles par un pointeur (le type pointeur est lié à la classe).

Au début de l'interprétation, la classe est vide.

A la déclaration on a défini le nombre maximum d'éléments qu'elle peut contenir. Les opérations permises sur une classe sont Alloc (empile) et Reset (dépile).

Le descripteur de la classe contiendra donc sa taille maximum et le type de ses éléments.

L'adresse d'un nouvel élément d'une classe est mis dans une variable de type "pointeur sur la classe".

La seule information du type pointeur est donc le type des éléments pointés.

II.2.d.8 Record

C'est une collection d'éléments, appelés champs, de types pouvant être différents.

De plus ces éléments peuvent avoir une existence conditionnelle s'ils sont dans un "CASE".

L'"instruction" CASE se fait sur un champ particulier appelé aiguillage (tagfield).

Plusieurs "CASE" peuvent être imbriqués (récursivité).

Les champs pouvant être de types différents, les alternatives du CASE (branche) peuvent avoir des tailles différentes. La variable de type "record" est donc de taille variable suivant les valeurs du 'tagfield'(ou des tagfields)(sélecteur de champ).

Nous voyons donc qu'un "record" peut être représenté par une arborescence dans laquelle un seul chemin est défini à un instant donné de l'interprétation. D'où la nécessité de décrire et de contrôler dynamiquement une telle structure.

Les champs doivent indiquer le type, la valeur et un moyen pour contrôler leur existence: accès à la branche décrivant l'alternative à laquelle ils appartiennent.

On peut avoir des classes de 'record', dans ce cas au cours de la procédure ALLOC, les valeurs des sélecteurs de champ seront précisées et la taille du 'record' calculée pour mettre à jour la zône libre dans la variable classe.

Exemple:

```
Déclaration PASCAL: R = record
                        A,B : integer ;
                        case T1 : 0..7 of
                            0,1 : (D,E : real) ;
                              2 : (case T2 : 0..3 of
                                        0 : (X : array(1..20)of char) ;
                                    1,2,3: (y : real) ;
                        4,5,6   : (F : 5..125) ;
                            7   : (G : integer ; I : boolean)
                        end ;
```

Structure logique globale:

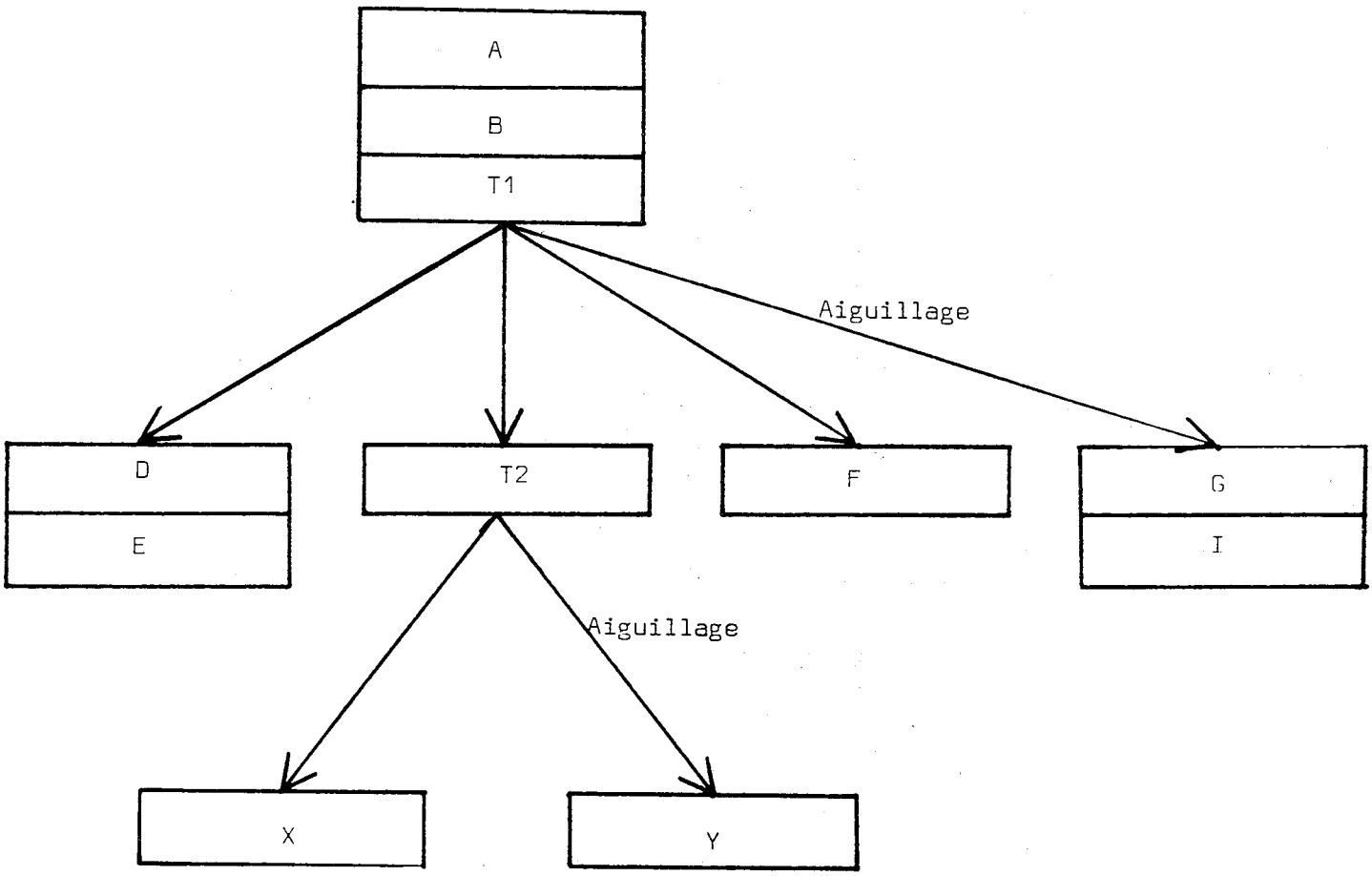


Figure II.1. Exemple de structure logique de 'record'

Représentation interne:

La représentation interne de la structure arborescente est alors:

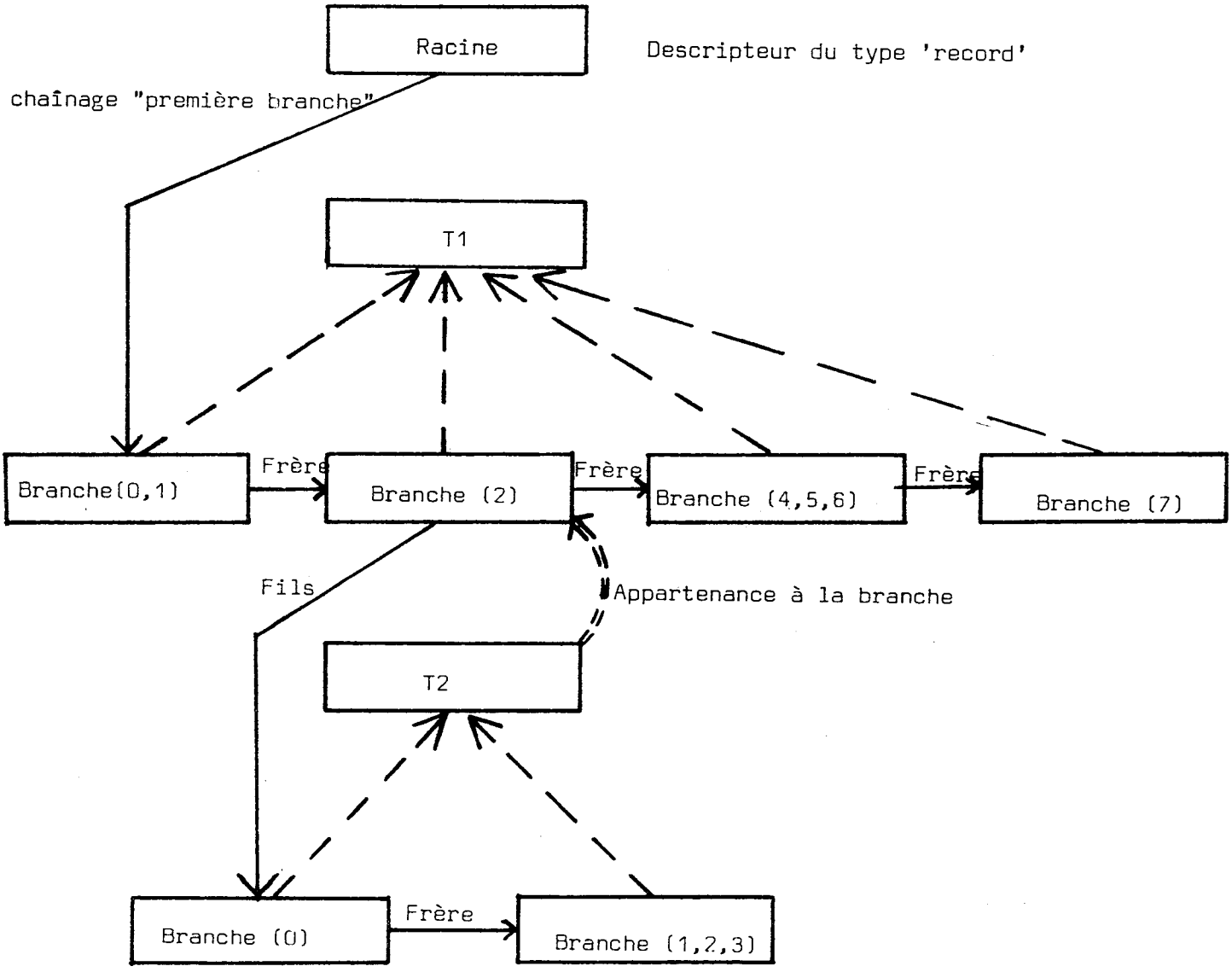


Figure II.2. Représentation interne du 'record'

Chaînage des branches (décrivant la structure conditionnelle de l'aiguillage).

- dans le cas de l'accès à un champ nous devrons vérifier son existence. C'est-à-dire que les tagfields dont il dépend ont bien les valeurs précisées dans la branche du champ.

D'où un chaînage vers le tagfield (Père) dans la branche (Remontée dans l'arbre: trait pointillé).

- dans le cas d'une allocation, nous allons pour le premier 'tagfield' chercher l'alternative vérifiée (branche), puis répéter l'opération pour les autres 'tagfields' emboîtés (descente dans l'arbre: trait plein).

II.2.e) Variables

Les variables ont le même nom interne et la même portée que les autres identificateurs déclarés.

A la déclaration, la seule information disponible est le type de cette variable. L'interprétation l'enrichira d'une valeur.

L'instruction de création a donc pour seul paramètre le nom du type de la variable. Le descripteur ainsi créé contient le nom interne du type ainsi qu'un moyen d'accès à la valeur dont la place a été réservée au cours du processus de création.

II.2.f) Procédures - fonctions

Les procédures et fonctions sont caractérisées à la déclaration par:

- nom,
- liste paramètres (peut être vide pour les procédures seulement)
  déclarations locales
  instructions
  ] définit un nouveau bloc.

Pour une fonction il y a en plus le type de la valeur retournée.

Le descripteur devra permettre de trouver toutes ces informations.

Proc Fonc	accès au type si fonction	niveau	variables locales	début des instructions	paramètres formels

Les paramètres formels de la procédure (ou de la fonction) auront eux aussi leur descripteur.

Le type du paramètre et le mode de passage sont précisés dans la déclaration de paramètre formel.

'Var'	passage par référence
{'Value'}	passage par valeur
'Procédure'	passage d'une procédure
'Fonction'	passage d'une fonction
{'Const'}	passage d'une valeur calculée une fois pour toutes à l'entrée de la procédure (affectation au paramètre formel interdit dans le corps de la procédure).

Nous appellerons ce dernier mode de passage de paramètre, passage par expression. La différence essentielle avec le passage par valeur provient du fait qu'il n'y a pas recopie pour les structures.

{ } indique que le mot clé est omis dans certaines versions de PASCAL.

Dans la déclaration du paramètre formel, nous avons spécifié:

- son type,
- son mode de passage.

Ce sont les deux renseignements à conserver dans le descripteur.

Remarques:

- les paramètres sont considérés comme des variables locales,
- dans une fonction l'affectation aux paramètres et aux variables non locales est interdite, (pas d'effet de bord).

II.2.g) Remarques complémentaires sur les déclarations

Tous les descripteurs possèdent un préfixe indiquant leur type. Dans le cas des variables, des informations complémentaires pour les opérateurs faciliteront l'interprétation future.

Ces informations permettront d'ailleurs de simplifier les descripteurs de variables dans les cas particuliers des variables de type prédéclaré.

Ce sont les principales contraintes d'économie qui influeront sur le deuxième pas d'obtention du langage intermédiaire.

II.3. INSTRUCTIONS

II.3.a) Généralités

Nous allons appliquer ici la méthode définie précédemment (I.5.d) au cas particulier du langage PASCAL.

Le choix entre la méthode procédurale ou la méthode opérative est guidé par l'étude des réalisations existantes de machines spécialisées pour un langage et les particularités du langage PASCAL.

II.3.b) Expressions - opérateurs

II.3.b.1 - Les opérateurs utilisés en PASCAL, rangés en priorité décroissante, sont les suivants:

opérateurs	opérandes	résultats	sémantique
¬	booléen	booléen	complémentation
*	entiers mixte(entier,réel) réels	entier réel réel	multiplication classique
/	entiers réels	entier réel	division classique
DIV	entiers	entier	division entière
MØD	entiers	entier	reste de la division entière
∧	powersets booléens	powerset booléen	ET logique
+	entiers réels	entier réel	addition classique
-	entiers réels	entier réel	soustraction classique ou - unaire
-	powersets	powerset	différence d'ensemble.
∨	powersets booléens	powerset booléen	OU logique
=, ≠	tout type sauf 'class' et 'file'	booléen	comparaisons classiques
<, >	tout scalaire	booléen	
≤, ≥	tout scalaire powersets	booléen	test inclusion d'ensembles
IN	scalaire et son powerset	booléen	

Remarque:

Dans la version IREP-CICG, l'opérateur '+', s'appliquant à des 'powerset', représente la disjonction.

Ces opérateurs ont les priorités classiques.

La méthode opérative sera employée comme dans les exemples connus. Nous aurons donc une mise en postfixé.

Le sens des opérateurs peut varier suivant les opérandes. Le '-' n'est pas le même pour des entiers et des powerset par exemple.

Pour éviter des lectures trop fréquentes de descripteur, il sera intéressant de prévoir un renseignement complémentaire pour l'opérateur dans le descripteur de variable (chapitre IV).

II.3.b.2 - Affectations

Avant l'affectation proprement dite, on contrôle que le type de la valeur de l'expression est bien compatible avec celui de la variable à affecter. Sinon il y a erreur.

Si le type de l'expression est un intervalle contenu dans le type de la variable, l'affectation est permise.

Une seule exception à la règle de type "égaux" pour l'affectation, est la conversion possible du type entier en type réel.

Comme les opérateurs classiques, l'affectation sera considérée opérativement, d'où postfixée.

II.3.c) Appel de fonction ou de procédure

Nous n'avons aucun contrôle à faire sur l'environnement lors de l'évaluation des paramètres (évaluation dans le contexte d'appel).

Nous considérerons l'appel de procédure comme une opération n-aire sur les valeurs des n paramètres effectifs (méthode opérative). Le réordonnancement est donc une mise en postfixé de l'appel.

Fonctions et procédures standards.

Certaines mériteront un traitement spécial. Les mesures statiques nous donneront les renseignements nécessaires.

nom	opérande	résultat	effet
Succ(x)	scalaire	scalaire	scalaire immédiatement supérieur à x
Pred (x)	scalaire	scalaire	scalaire immédiatement inférieur à x
Abs (x)	entier ou réel	entier ou réel	valeur absolue
Sqr (x)	"	"	carré de x
Odd (x)	entier	booléen	parité
Trunc (x)	réel	réel	valeur tronquée
Int (x)	caractère	entier	conversion:caractère-entier
Chr (x)	entier	caractère	conversion:entier-caractère
Eof (x)	fichier	booléen	vrai si 'fin de fichier'
Alloc (P)	pointeur		voir § I.2.d (class et record
Alloc(P,t_1...t_n)			
Reset (P)	pointeur		
Reset (P)	fichier		retour au début du fichier
Get (f)	fichier		lecture de l'enregistrement suivant
Put (f)	fichier		écriture de l'enregistrement suivant
Weorf (f)	fichier		écriture d'une fin d'enregistrement

II.3.d) Instructions d'accès

La rencontre d'un identificateur sera traduite par l'accès à son descripteur (référence au nom intern)

NOM S, D

S information relative au bloc

D information relative à la déclaration dans le bloc.

Pour accèder à un élément de structure il existe trois modes d'accès suivant le type de la structure (pouvant être combinés):

- indexation de tableau,
- accès à un champ de "record",
- indirection sur un pointeur.

Aucun environnement n'étant nécessaire, la méthode opérative est employée.

D'où:

indexation:

|<exp$_1$>	[
:	<exp$_1$>
	,
|<exp$_n$>	:
	<exp$_n$>
INDEX n	]

Dans le cas d'un pointeur le postfixé et l'infixé sont équivalents.

↑...↑ sera traduit par PØINT n-1

n flèches consécutives

Pour les 'records' le postfixé donnera:

NOM S, D	•
CHAMP	<ident>

Ces deux instructions apparaissent toujours pour l'accès à un champ. D'où une contraction immédiate en une seule instruction :

CHAMP S, D

Remarques:

- Dans une instruction "With" un champ peut être nommé directement. Deux solutions:

  . remettre à la traduction le nom de la variable Record (With traité à la traduction) ;

  . traiter le "with" à l'interprétation. Nous avons alors une instruction spéciale CHAMW I,j ; S,D
  Var.Record

- Dans le cas où une variable apparaît en partie gauche, seule l'adresse de la valeur est importante. D'où une instruction IDENT S,D qui recherch cette valeur (en effet, dans le cas de variable simple, l''information su la valeur' dans le descripteur, peut être la valeur elle-même.

II.3.e) Remarques sur les instructions composées:

Contrairement aux instructions traitées jusqu'ici, nous avons besoin d'un contrôle.

Le saut à l'intérieur d'une boucle, ou d'une alternative, étant permis (puisque non interdit) il est intéressant de le détecter. En effet, dans le cas d'une boucle 'for', l'initialisation n'a pas été faite pour le 'with', il n'y a donc pas eu calcul du nouvel adressage, et, pour les instructions de choix, les tests n'ont pas été faits.

Toutes ces situations aboutissent à un résultat indéfini qui doit être détecté. Nous utiliserons donc ici la méthode procédurale pour ce contrôle: création d'un descripteur de segment pour détecter les situations décrites plus haut. Dans la deuxième phase, une simplification pourra être faite.

II.3.f) Instruction de choix

Nous avons vu au chapitre précédent le traitement procédural de l'instruction 'if'.

En rebaptisant l'instruction 'PVG' par 'SORT' qui est plus explicite et 'else' par 'FIN', nous obtenons:

```
  if                IF            création d'un descripteur de segment 'if'
                    |<exp>
<exp>
                     THEN         test et branchement éventuel
 then
                    |<inst 1>
<inst 1>
                     FIN
 else
                    |<inst 2>
<inst 2>

  ;                 SORT          fin, "dépilage" du descripteur de segment'IF'
```

Remarque:

L'instruction de séquentialité 'PVG' est omise ici en PASCAL lorsqu'elle provoque seulement le passage à l'instruction suivante, sans manipulation de descripteur de segment ni branchement.

Instruction aiguillage ("CASE"):

case <exp> of e_1: <inst 1>; .... e_n: <$inst_n$> end ;

La méthode procédurale donne de même, après sortie du paramètre par valeur <exp>:

case		CASE	création d'un descripteur de segment 'CASE'
<exp> of		|<exp>	
e_1:		OF	début 1ère alternative
		e_1	liste valeur
<inst 1>		DPT	test et branchement si faux
		|<inst 1>	
i	environnement 'CASE'	FIN	
:		:	
e_n:		e_n	
		DPT	
<inst n>		|<inst n>	
end ;		SORT	fin du segment CASE

Nous pouvons aussi considérer que les littéraux indexant les instructions, sont des déclarations du corps de la procédure.

L'écriture formelle est alors, en factorisant les 'deux points':

CASE (<exp> of $<e_1>$,...,$<e_n>$,:, $<inst_1>$;...$<inst_n>$);

```
D'où    CASE                    création descripteur segment 'case'
          <exp>
        OF                      mode 'déclaration'
          <e1>                  création tableau de correspondance
                                       étiquette   adresse
          <en>
        DPT                     branchement indexé
        |<inst1>
        FIN                     "exit du segment"
        :
        |<instn>
        SORT                    fin, "dépilage" segment 'case'
```

Nous retrouvons le principe du tableau de correspondance dans la compilation de cette instruction.

Les mesures statiques et dynamiques nous permettront de choisir et d'améliorer une de ces deux solutions.

II.3.g) Instruction répétitive

while exp do inst ;

la traduction produit:

while		WHILE	création du descripteur de segment 'while'
<exp>		|<exp>	
do	segment 'while'	DO	test et sortie éventuelle du segment
<inst>		|<inst>	
;		SORT	dans cet environnement provoque le bouclage

Repeat <instlist> until <exp>;

REPEAT (<instlist> until <exp>;

'until' est un opérateur de séquencement qui a comme opérande l'expression. D'où le post-fixage à l'intérieur du "corps" de la procédure de cet opérateur.

REPEAT	création du descripteur de segment Repeat
|<instlist>	
|<exp>	
UNTIL	test et bouclage éventuel
SORT	fin, sortie de l'environnement

- FOR <var> := $<exp_1>$ to $<exp_2>$ do <inst>;
  downto

Remarques:

- le pas de l'instruction FOR est implicite, +1 dans le cas 'To' ou -1 pour 'downto'.

- la variable ('ident') doit être déclarée et de même type que <exp_1> et <exp_2>
- et à la sortie de la boucle la valeur de la variable de contrôle est considérée comme indéfinie.
- les valeurs de <exp_1> et <exp_2> sont calculées une fois pour toutes à l'entrée de la boucle.

FOR (<var> := exp_1 { to / downto } <exp_2> do <inst>) ;

Sémantiquement 'to' ('downto') a deux significations complémentaires:
- boucle croissante (décroissante)
- incrémentation (décrémentation) de la variable de contrôle.

La première signification est une particularisation de la boucle, on peut s'en servir pour particulariser l'environnement.

D'où deux instructions: 'FORUP', 'FORDOWN'.

FOR	FORUP(DOWN)	création descripteur boucle croissante (décroissante)
<var>	|<exp_1>	
:=	|<exp_2>	
<exp_1>		
to	|<var>	accès à l'adresse puisque dans environnement 'boucle for'
<exp_2>	AFFECT	(post-fixage de l'affectation)
do	TO (DOWN)	test et sortie éventuelle du segment
<inst>	|<inst>	
;	SORT	incrément (décrément) et bouclage.

II.3.h) Instruction 'with'

with <varlist> do <inst>;

Les variables de la liste précisée dans le 'with' sont de type 'record'. Le 'with' permet de ne plus avoir à préciser les noms de ces variables pour accéder à un champ particulier, le nom de ce champ étant suffisant.

C'est une facilité d'écriture offerte au programmeur. De plus l'adresse de ces variables est calculée une fois pour toutes à l'entrée dans le 'with'.

Nous avons ici l'ouverture d'un environnement d'adressage et de contrôle

WITH	WITH	entrée dans le nouvel environnement d'adressage
<varlist>	|<varlist>}	"calcul des paramètres"
do	DO	empilage des descripteurs des "opérandes"
<inst>	|<inst>	
;	SORT	

II.3.i) Instruction begin-end

Aucune déclaration n'est permise à l'intérieur d'un groupe begin-end. Son seul effet est de ramener syntaxiquement une liste d'instructions à une seule <instruction> du langage source.

Cette instruction sera donc ignorée.

Begin-end d'une procédure:

- BEGIN marque le début de la procédure. C'est le passage du mode déclaration au mode instruction.

- END marque la fin de la procédure. Il sera traduit par une instructior "RETOUR" qui effectuera le retour dans le contexte d'appel.
Cette instruction est la même pour les procédures ou les fonctions.

III.- MESURES STATIQUES

III. 1- INTRODUCTION

Ces mesures ont été faites sur un lot de programmes venant de l'Université d'AARHUS (Danemark) et sur le compilateur PASCAL écrit en PASCAL (ZURICH).

En tout 20.000 lignes de PASCAL ont été testées.

La taille des programmes varie entre 25 et 4.700 lignes (dans le cas du compilateur de ZURICH).

Les vingt-deux programmes PASCAL se répartissent en :

- 18 programmes de moins de 200 lignes,
- 1 programme de 900 lignes,
- 1 programme de 1.140 lignes,
- 1 programme de 2.000 lignes,
- 1 programme de 4.700 lignes.

Du point de vue de la représentativité, nous pouvons remarquer que:

- l'éventail des tailles est suffisant,
- par contre le nombre de programmes de taille moyenne est un peu faible.

Seules n'ont pas été comptées les occurences statiques des opérateurs. Pour ceux-ci, des mesures dynamiques,seules,nous paraissent intéressantes. Les résultats sur les déclarations vont nous guider dans le choix des formats et des codages pour les descripteurs. Ceux sur les instructions vont nous permettre de définir les codages minimaux des instructions. Ces mesures nous donnent des indications pour minimiser l'encombrement mémoire.

Les mesures dynamiques nous permettront de minimiser le temps d'exécution Ces dernières seront faites en insérant des fonctions de comptage dans la première version de l'interpréteur.

III.2. DECLARATIONS

- Occurence: nombre de fois rencontrée

nom	occurences totales
étiquettes externes	29
constantes	116
variables	1767
types	901
définition de type	75
REFERENCE TYPE	1284

- Pourcentage par niveau lexicographique:

NOM	au nivau global(1) en %	au niveau 2 en %	niveau supérieur à 2	TOTAL
étiquette externe	56	41	3	100
constantes	97,6	1,6	0,8	100
variables	44	51	5	100
types	78,5	20,1	1,4	100

Remarques:

- les types déclarés représentent 8,3 % des types utilisés.
- la moyenne d'utilisation d'un type est 2, ceci étant dû à la très grande utilisation des types pré-déclarés comme "integer" ou "real" et la déclaration implicite fréquente pour les structures.

On constate que les constantes sont presque toutes déclarées au niveau global. On ne perd donc pas beaucoup de place en les mémorisant au

niveau global. Ceci sera fait pour les constantes de grande taille (constante chaîne de caractères par exemple). Les autres seront simplement considérées comme des synonymes de la valeur qu'elles représentent et nous utiliserons le littéral correspondant.

- La plus grande partie des types est aussi globale.

- Les types sont parfaitement statiques et peuvent être entièrement calculés à la traduction.

- Même au cours d'appel récursif d'une procédure, une seule occurence de ces descripteurs de type est nécessaire.

Ces trois considérations nous amènent à ranger tous les types dans une table, sans mécanisme de pile. D'où un nom interne réduit à un index dans la table. On évite ainsi les instructions de gestion et de construction des types, regagnant par là la place mémoire perdue par l'utilisation de la table.

Une conséquence immédiate est la transformation des instructions d'accès à un champ:

CHAMP S,D devient CHAMP p(p: index dans la table des descripteurs de type)

CHAMW i, j; S,D devient CHAMW i, j;p

III.3. DECLARATION DE TYPES

nom	occurences totales	pourcentage relatif (en %)
pointeur	34	3,8
"powerset"	7	0,8
"record"	96	10,6
fichier	12	1,3
classe	30	3,3
tableau	280	31,1
scalaire	24	2,7
intervalle	418	46,4
total	901	100,0

Remarques:

- La plupart des types "intervalles" se trouvent en fait dans les déclarations de tableau pour en définir les bornes.

- Les types intervalles restants (70 à 100 occurences) ne sont que des intervalles d'entier (contrainte d'implémentation du langage PASCAL).

L'emploi très faible du type "powerset" (0,8 %) semble justifier l'abandon d'un descripteur particulier. Le descripteur de l'intervalle sur lequel est bâti le "powerset" sera suffisant.

Le type le plus fréquent est le tableau avec un (ou plusieurs) intervalle associé (index). Un gain de place dans le descripteur sera donc très intéressant.

Nous remarquons que le "pas" est répété deux fois dans le "Dope-Vector". Si nous supprimons cette répétition, nous obtiendrons un Dope-Vector plus court.

Les types les plus fréquents sont ensuite les intervalles seuls (0,11%) et les "records" (10,6%). Pour les "records" peu d'optimisation est possible. Par contre dans le cas des intervalles, on constate que, comme pour les littéraux (cf.§ 6), les valeurs rencontrées sont faibles. Un nouveau type prédéclaré ("entier court") nous parait donc souhaitable pour les entiers.

Renseignements complémentaires:

nom	occurences totales	moyenne/record
nombre aiguillage	42	0,44
nombre champ	553	5,77

Moyenne de branches dans un **aiguillage:** 2,7 (alternative dans l'aiguillage: "CASE").

Finalement, moins de 44% des record possèdent un aiguillage: le chaînage de branches ne sert à rien dans les autres cas.

Une autre version des descripteurs de champ permettrait de ne pas perdre cette place.

nom	moyenne	maximum
dimension de tableau	1,2	2
nombre d'ident. dans un scalaire	4,03	8

Apparamment un scalaire de 4 bits suffirait pour tous les scalaires définis par le programmeur, mais ce serait une limitation trop forte. D'autre part, nous n'avons pas intérêt à trop multiplier les types, sous peine de perdre en complexité ce que nous gagnons en place.

III.4. PROCEDURES ET FONCTIONS

III.4.a) Procédures et fonctions

nom	nbre déclaré	pourcentage relatif	nbre d'appels (prédéclarés compris)	nbre moyen d'appels d'une procédure
Fonction (+ prédéclarés)	24 (+15)	7	837	21,4
Procédure (+prédéclarés)	178 (+12)	52,4	2776	14,6
Procédure sans paramètres	138	40,6	615	4,5
TOTAL	340(+27)	100	4228	

La différence du nombre moyen d'appels entre procédure avec paramètre et sans paramètre s'explique en partie par la présence de procédures prédéclarées très utilisées dans le premier cas ('get' et 'put' notamment).

nombre d'appels à des fonctions ou procédures prédéclarées	2198	52 %
nombre d'appels à des fonctions ou procédures non prédéclarées	2030	48 %
Total	4228	100 %

Remarque:

Si on enlève les procédures prédéclarées, on trouve que le nombre moyen d'appels des fonctions ou procédures avec paramètre est de 7, nettement supérieur à celui des procédures sans paramètre (4,5).

On constate donc que 52 % des appels de procédure ou fonction se fait sur des prédéclarées. D'où l'importance de minimiser le travail à accomplir lors de ces appels. Pour la plupart, les fonctions ainsi appelées sont très simples (ex.: successeur (x), valeur absolue (x)...)

Une accélération est de remplacer l'appel d'une fonction simple par une instruction du code intermédiaire la représentant.

Les procédures prédéclarées se divisent en deux catégories:

- les procédures d'entrée-sortie: get, read, write, put...
- les autres: Alloc, Reset..., Append...

Remarque:

Le scalaire "character" prédéfini a 64 valeurs possibles, ce qui implique six bits pour le codage (le CDC est à mots de 10x6 bits).

Avec une valeur de scalaire sur 7 bits, nous recouvrons largement toutes les possibilités et pouvons introduire de nouveaux caractères, par exemple le '"' ou le '@' qui n'existent pas sur le CDC.

Pour les procédures les plus utilisées, en dehors de celles d'entrée-sortie, une transformation similaire devra être faite.

Nous aurons donc trois catégories de traitement à la traduction des fonctions et procédures prédéclarées.

- les entrées-sorties: (get, put, eof, read, write, reset)
  appel à des routines système

- fonctions simples et procédures de gestion de class: (suc, pred, abs, ..., alloc, reset).
  Une ou plusieurs instructions du langage intermédiaire.
- les autres: sin, cos, insert, exp,...
  inchangées.

III.4.b) Paramètres

type	occurences totales	pourcentage relatif en %
par valeur	2	0.6
par référence	77	21,6
par expression	273	76,7
procédure	0	0
fonction	4	1,1
total	356	100,0

Remarque:

En ce qui concerne l'appel par expression, il a été supprimé dans la nouvelle version du langage. Seul subsiste l'appel par valeur et celui par référence. Dans ces conditions, les paramètres par valeur représentent 77,3 % des paramètres.

La construction d'un descripteur d'indirection n'a lieu que dans 22,7 % des cas.

III.5. INSTRUCTIONS D'ACCES

nom	occurences totales	pourcentage relatif en%	commentaires
indexation	1923	10,3	indexation d'un tableau
champ	1021	5,5	accès à un élément de variable de type record.
champwith	1498	8	même chose que ci-dessus mais la variable est dans un with.
flèche	642	3,2	pointeur dans une class élément de fichier.
ident	13643	73	
total	18727	100,0	

Comme les déclarations pouvaient le laisser prévoir (31,1 % de type de tableau), les indexations sont les instructions d'accès à des éléments de structure les plus nombreuses.

Le fort pourcentage du 'champwith' par rapport à l'instruction 'champ' normale est due en partie à l'emploi général du 'with' dans le compilateur. L'utilisation de cette instruction PASCAL est moins fréquente dans les autres programmes (nouveauté de ce genre d'instruction...)

L'économie d'instruction d'accès au tableau, pour chaque indexation sera non négligeable. Pour cela il suffit d'une deuxième instruction INDEX:

```
NOM S,D   }   devient INDEX S, D; n
INDEX n
```

(économie de place, mais non de temps d'exécution)

Nous constatons que les accès aux champs d'une variable 'record' sont plus nombreux dans un 'with' qu'ailleurs.

On se propose de conserver à l'interprétation le mécanisme du with: le calcul de l'adresse de la variable 'record' est fait une fois pour toutes à l'entrée dans le 'with'.

(Ceci est intéressant si la variable en question est elle-même un élément de structure. Exemple: A[I,J]↑.B[k]). Les indexations, accès au champ et indirections sont effectués une seule fois à l'entrée dans le WITH).

On économise alors, en temps d'exécution et en place, au prix d'une instruction 'champ' spéciale et d'un adressage particulier.

III.6. LITTERAUX

III.6.a) Caractères et entiers

nom	occurences totales	pourcentage relatif en%	maximum d'éléments	nombre moyen
Littéral caractère ou chaîne de caractères	2029	29,1	57	/
Litteral powerset	68	0,9	11	2,28
Litteral entier	5056	70	/	/

Remarque: Les littéraux réels et booléens ne sont pas comptés.

Le nombre de littéraux "powerset" est presque négligeable (0,9 %) devant les autres (entiers 70 %, caractères ou chaînes 29,1 %). Nous n'aurons donc qu'un seul format de tels littéraux.

Il y aura deux littéraux distincts: pour les caractères (scalaire) et les chaînes de caractères (tableau).

III.6.b) Littéraux entiers

Nota:
La valeur est une valeur absolue. La valeur relative occupe n+1 bits (sauf pour les valeurs 0 ou 1).

valeur	occurences totales	pourcentage relatif en%
0	1264	24,3
1	1354	26,1
$1 < \quad < 2^3$	1158	24,6
$2^3 < \quad < 2^7$	1280	22,3
$2^7 < \quad < 2^{11}$	87	1,67
$2^{11} < \quad < 2^{15}$	20	0,38
plus de 2^{15}	34	0,65
Total	5197	100,0

On peut remarquer que 97,3 % des littéraux entiers tiennent sur 8 bits Comme nous l'avons déjà dit au § 3, nous introduisons un nouveau type entier: entier court en demi-format (12 bits de valeur). Dans ce cas 99 % des littéraux tiendront dans cet entier court.

On constate aussi que parmi ces littéraux entiers, 50,4 % sont 0 ou 1. De plus, nous aurons le littéral booléen 0 ("false"), le littéral adresse 0 ("NIL") ou le littéral booléen 1 ("true").

Remarques:

- le littéral 0 est utilisé le plus souvent dans [initialisation, comparaison]

- le littéral 1 est utilisé le plus souvent dans [boucle for, comme incrément: (P:=P+1)]

L'utilisation possible de la fonction "Succ" pour des entiers supprimerait une grande partie des littéraux 1, mais elle n'est pas employée par le programmeur. P := succ(P) ⟷ P := P+1. Ce pourrait être une optimisation faite par le traducteur.

Nous pouvons remarquer qu'il existe dans le B 6700 des instructions "littéral 0" et "littéral 1".

Dans cette optique, nous introduirons une nouvelle instruction:

UNZER P, Φ

P : préfixe
- PT : pointeur
- E : entier
- B : booléen

Φ : 0 ou 1

III.7. INSTRUCTIONS

III.7.a) Résultats globaux

noms	occurences totales	pourcentage relatif en%
if (sans else)	1067	11,4
if (avec else)	954	10,1
while	203	2,2
repeat	207	2,3
with	371	4
for	379	4,1
goto	304	3,2
exit	63	0,7
affectation	5694	60,7
case	120	1,3
total	9354	100,0

nombre de cas dans un "CASE"	
moyen	maximum
3,3	30

Si l'on s'en tient aux seules instructions de contrôle, les deux formes du if (21,5 %) représentent plus que toutes les autres instructions de contrôle (17,8 %). D'autre part, les deux formes du 'if' sont à peu près équilibrées (47 % avec else, 53 % sans else).

Nous distinguerons deux instructions: IF - IFELSE.

Remarque:

Les mesures réalisées par WORTMAN [WO-72] et WICHMANN [WIC-60] montrent aussi que le IF est l'instrcution composée la plus utilisée.

A l'inverse les instructions les moins employées sont EXIT et le CASE (respectivement 0,7 et 1,3 %). Il est intéressant de vérifier que cette tendance reste, au cours des mesures dynamiques (§ 6).

III.7.b) Affectation

Nous avons compté séparément les affectations à une variable de celles à un élément de structure.

nom	occurences totales	pourcentage relatif
affectation à 1 var simple	4694	82,5 %
affectation à 1 élément de struct.	1000	17,5 %

On constate que 82,5 % des affectations sont des affectations simples et seulement 17,5 % se font à des éléments de structure.

Comme pour l'indexation, nous proposons d'avoir deux instructions "AFFECT". Dans 82,5 % des cas, les deux instructions successives IDENT S,D (calcul adresse), AFFECT(affectation) seront contractées en une seule AFFECT S,D , d'où un gain important en place et aussi en vitesse d'exécution.

III.8. REFERENCES

- aux variables, procédures ou fonctions:

référence	occurences totales	pourcentage relatif en%
Prédéfini (niveau 0)	2198	9,1
Local (niveau courant)	6852	28,5
Global (niveau 1)	11742	48,7
Paramètres	1114	4,6
Local=global (dans niveau 1)	1951	8,1
Intermédiaire (autres niveaux)	208	0,9
total	24065	100,0

local = global: quand le niveau local est le même que le niveau global.

Dans PASCAL, les paramètres sont considérés comme des variables locales à la procédure.

Nous remarquons que 99,1 % des références se font aux niveaux "local", "global" et "prédéfini".

Le niveau global étant le plus privilégié (56,8 %) suivi de près par le niveau local (33,1 %), une implémentation du nom interne S (niveau), D (numéro de déclaration) pourrait être réalisée avec seulement 2 bits.

Par exemple:

$$S = \begin{cases} 00 & : \text{prédéfini} \\ 01 & : \text{global} \\ 10 & : \text{local} \\ 11 \;||\; (\text{n° niveau}) & : \text{intermédiaire} \end{cases}$$

99,1% des références

Il est de plus, inutile d'utiliser un "Display" pour seulement o,9 % des références. Dans ce cas un chaînage statique est suffisant.

Remarque 1:

Seulement 25% des programmes examinés avaient des procédures de niveau supérieur à 2 (références intermédiaires possibles).

Remarque 2:

Au niveau de l'interprétation, les seuls "prédéfinis" existants sont les fonctions ou procédures prédéclarées (voir § 4).

III.9. INITIALISATIONS

type	occurences totales	pourcentage relatif
variable	139	57 %
tableau	105	43 %
total	244	100 %

Les initialisations de tableau sont importantes. Un mécanisme spécial pourrait accélérer cette initialisation et surtout réduirait le nombre d'instructions nécessaires. Dans la mesure du possible, il faudrait garder la simplification d'écriture au niveau du code généré.

Exemple: $A = (vi_1, 25 * vi_2, \text{---} v_{in})$;

NOM S, D

Init v_{i1}, 1

Init v_{i2}, 25

⋮

Init v_{in}

Remarque:

L'initialisation ne fait pas partie du langage PASCAL. C'est une possibilité offerte au programmeur dans une implémentation particulière.

Elle est, dans les compilateurs actuels, réservée au niveau global.

III.10. MESURES PAR NIVEAUX LEXICOGRAPHIQUES (voir figures III.3 à III.8

Les courbes des maximums nous donneront une idée des tailles des noms internes et de la taille de la table pour les types.

Nous aurons alors le format complet des descripteurs et grâce aux mesures sur les types, nous pourrons estimer la taille moyenne d'un descripteur de type.

Les variables et procédures sont renommées par une information sur le bloc et un numéro de déclaration.

Comme pour les types, nous calculerons la taille moyenne de leurs descripteurs et nous en déduirons la taille maximum nécessaire aux différents niveaux de ce numéro.

On peut envisager, comme le fait WORTMANN, deux versions du numéro de déclaration:

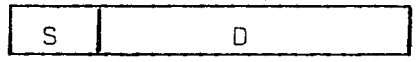
D = Flag || Num.

$$\text{Flag} = \begin{cases} 0 : \text{numéro de 4 bits} \\ 1 : \text{numéro de 12 bits} \end{cases}$$

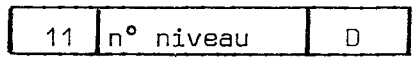
Le rôle de "Flag" pourra être tenu par le codage des niveaux lexicographiques (cf. § 8).

Le cas variable intermédiaire ne pouvant se produire qu'à partir du niveau 2, nous pourrons récupérer une partie de la place du numéro de déclaration pour un numéro de niveau. La taille de l'adresse (nom interne) reste alors constante, d'où un décodage facilité.

Exemple:



pour S = 00, 01, 10



S

dans cas intermédiaire

Le numéro de déclaration plus court n'est pas grave puisque ceci ne peut se produire qu'à partir du niveau 2 où les déclarations sont beaucoup moins nombreuses.

III.11. CONCLUSION

Comme nous l'avons vu au chapitre précédent, toutes les mesures n'ont pas été exploitées ici.

Dans le chapitre suivant, nous définirons complètement les descripteur et le code intermédiaire. Nous pourrons alors déterminer les tailles moyennes des descripteurs et des instructions (dans le cadre de la réalisation fine). Nous pourrons donc estimer les tailles des adresse de ces différents objets.

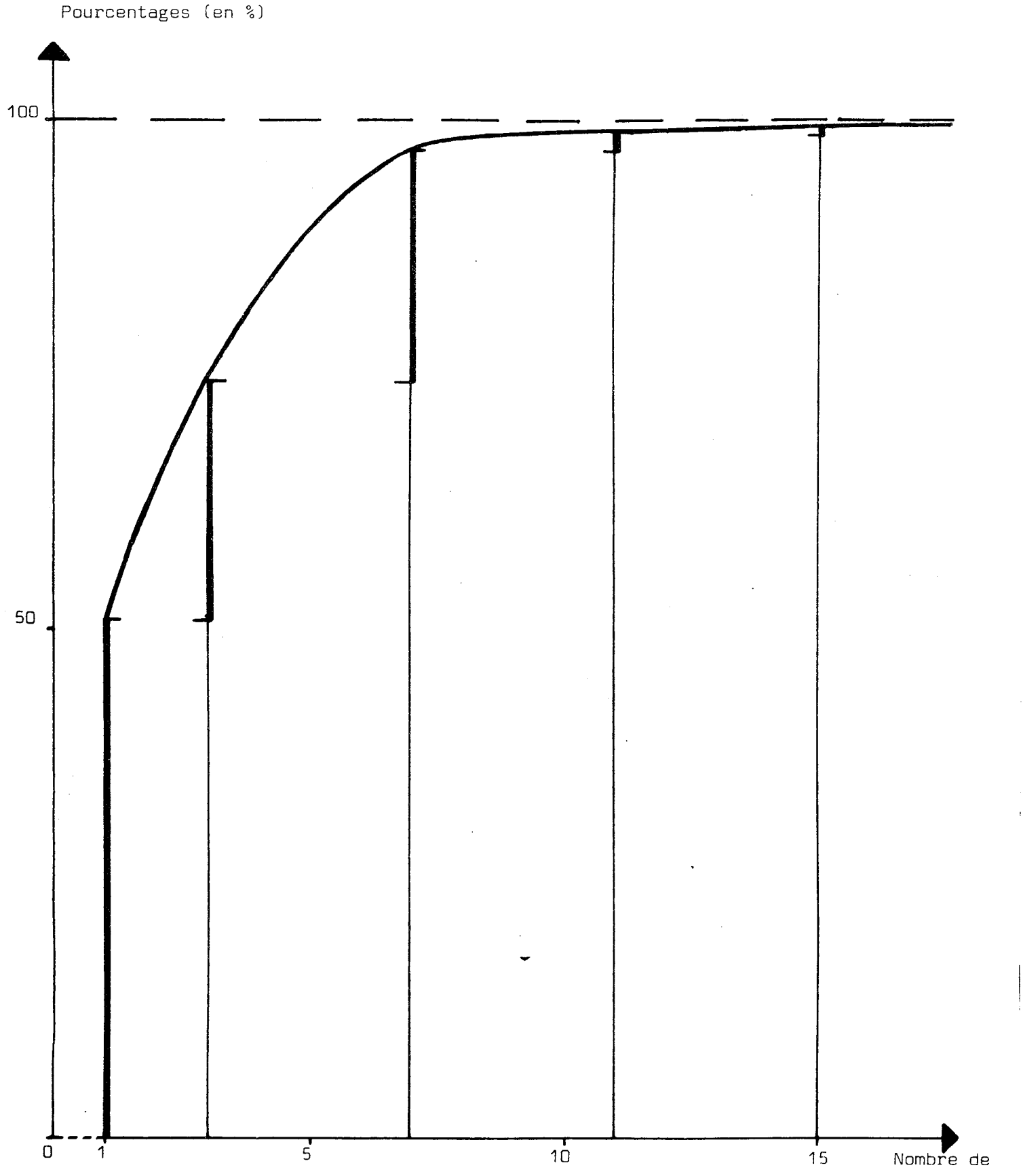


Figure III.1. Répartition des littéraux entiers

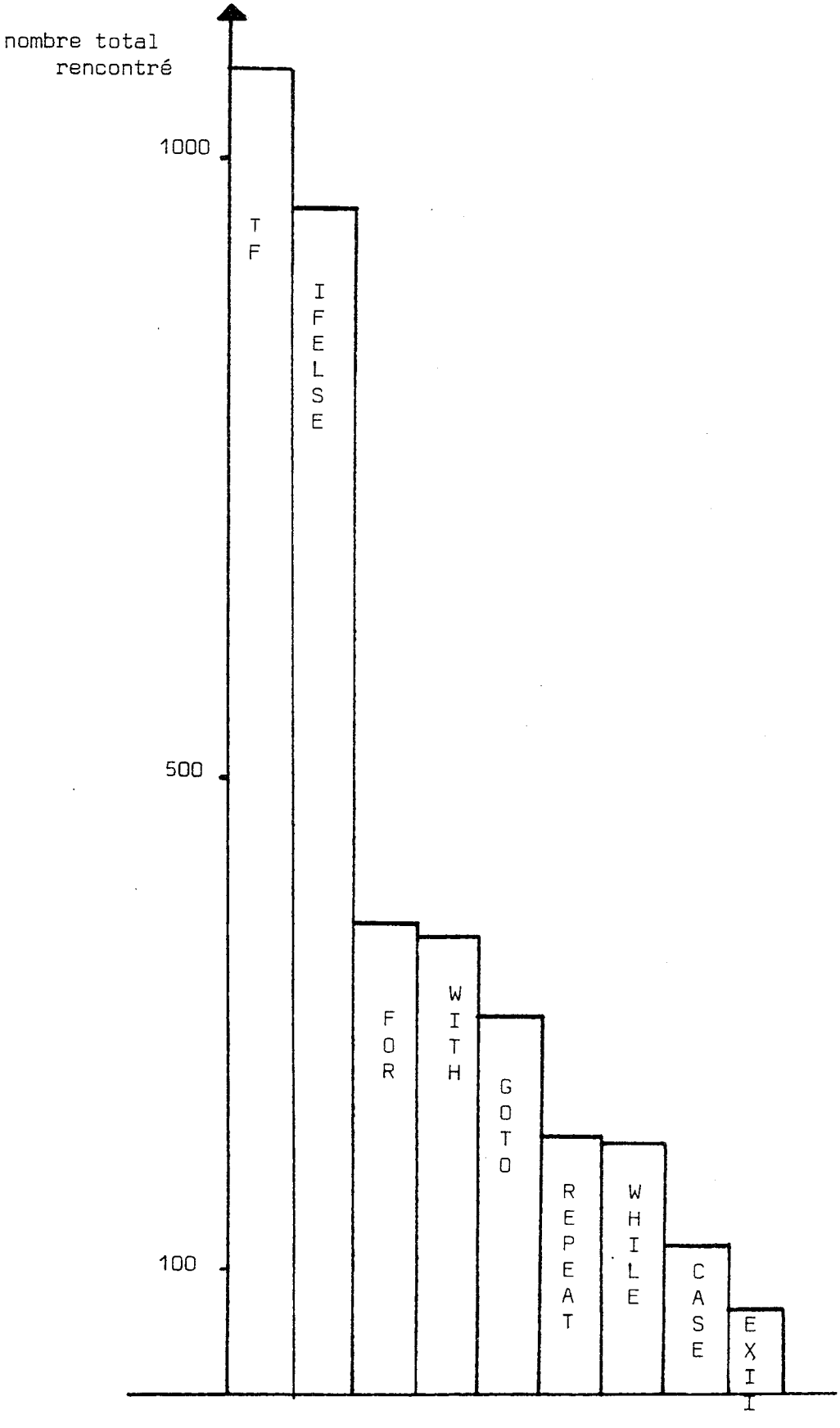


Figure III.2. Classification des instructions de contrôle

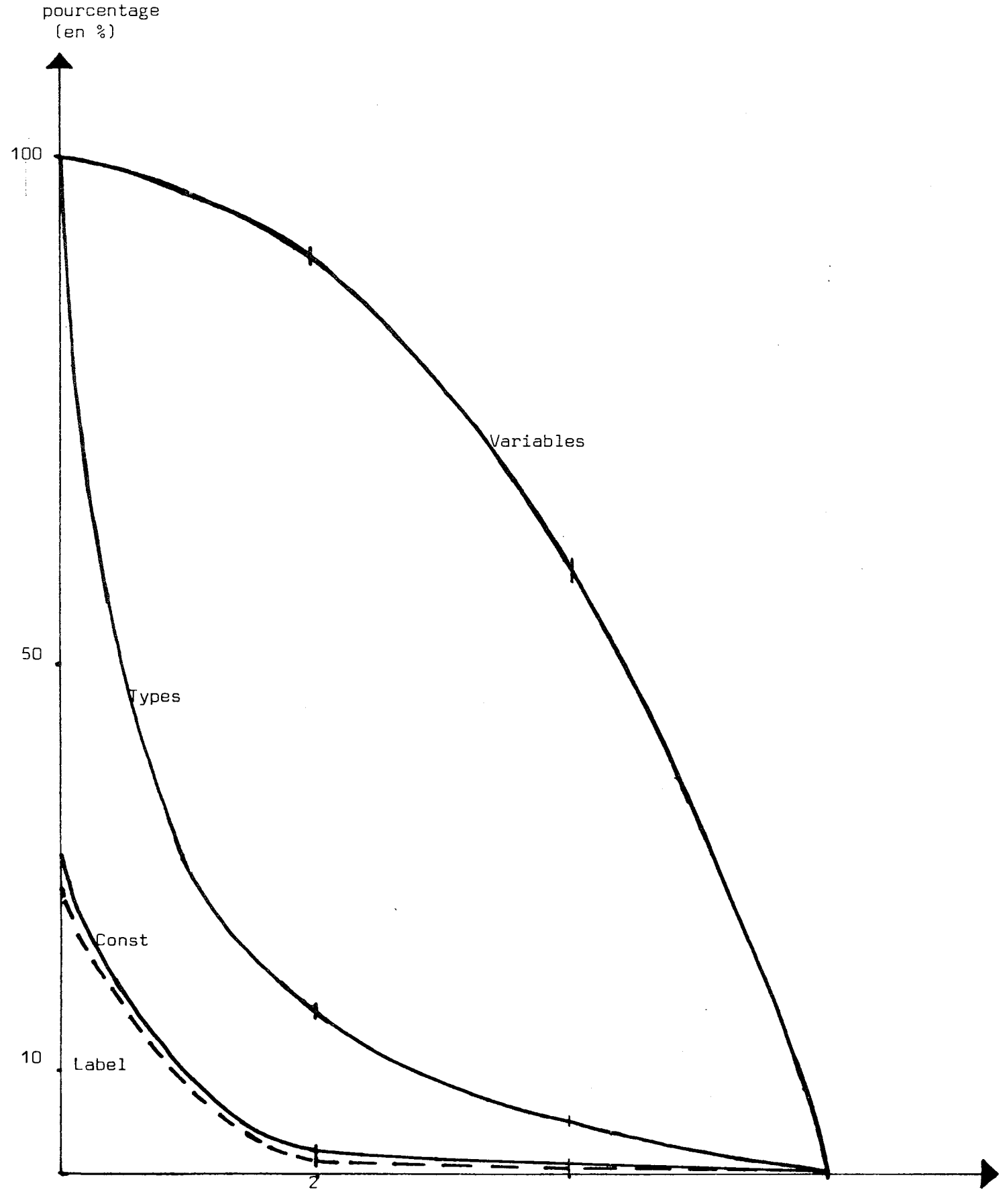


Figure III.3. Répartition par niveau lexicographique des déclarations

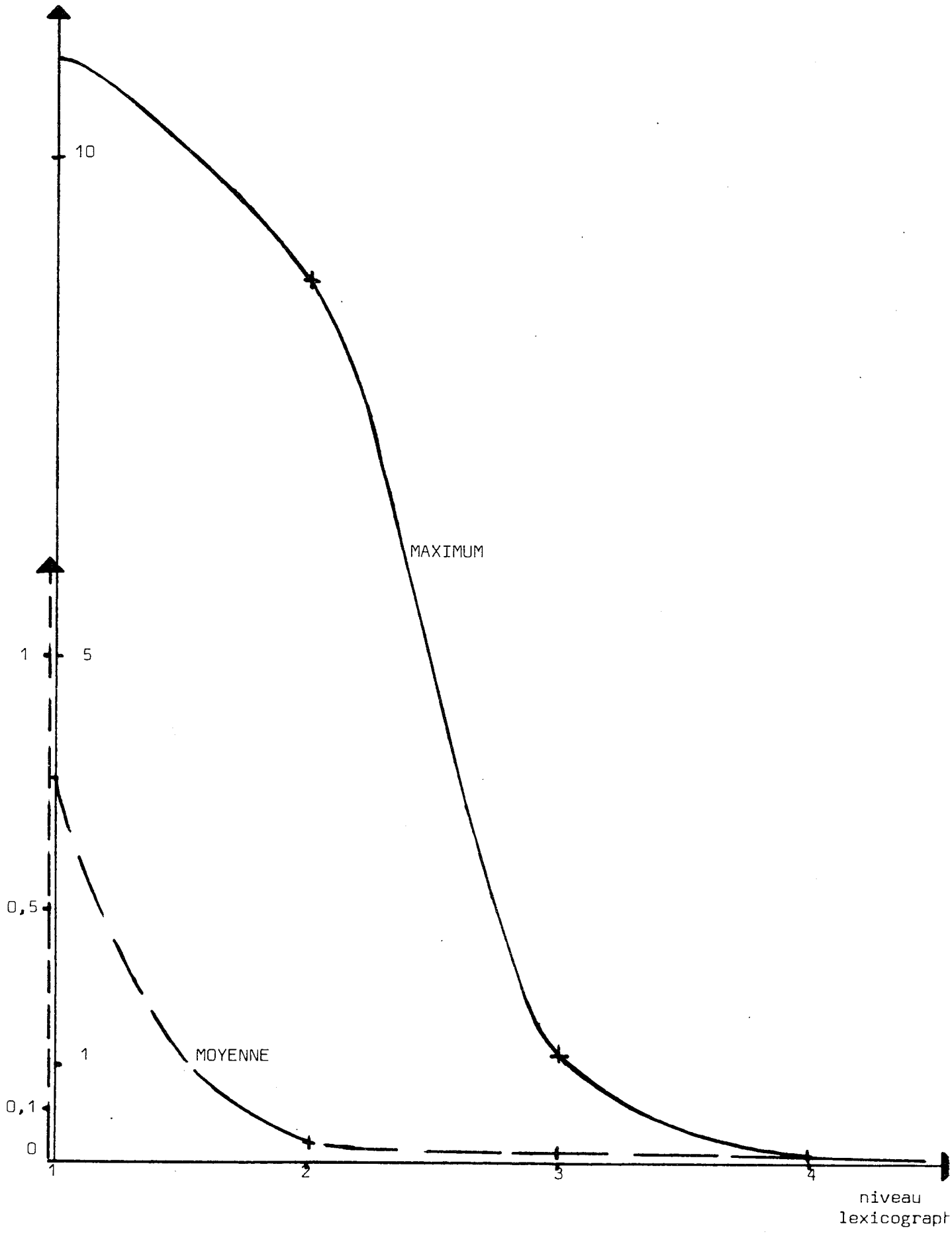


Figure III.4. Déclarations d'étiquettes externes

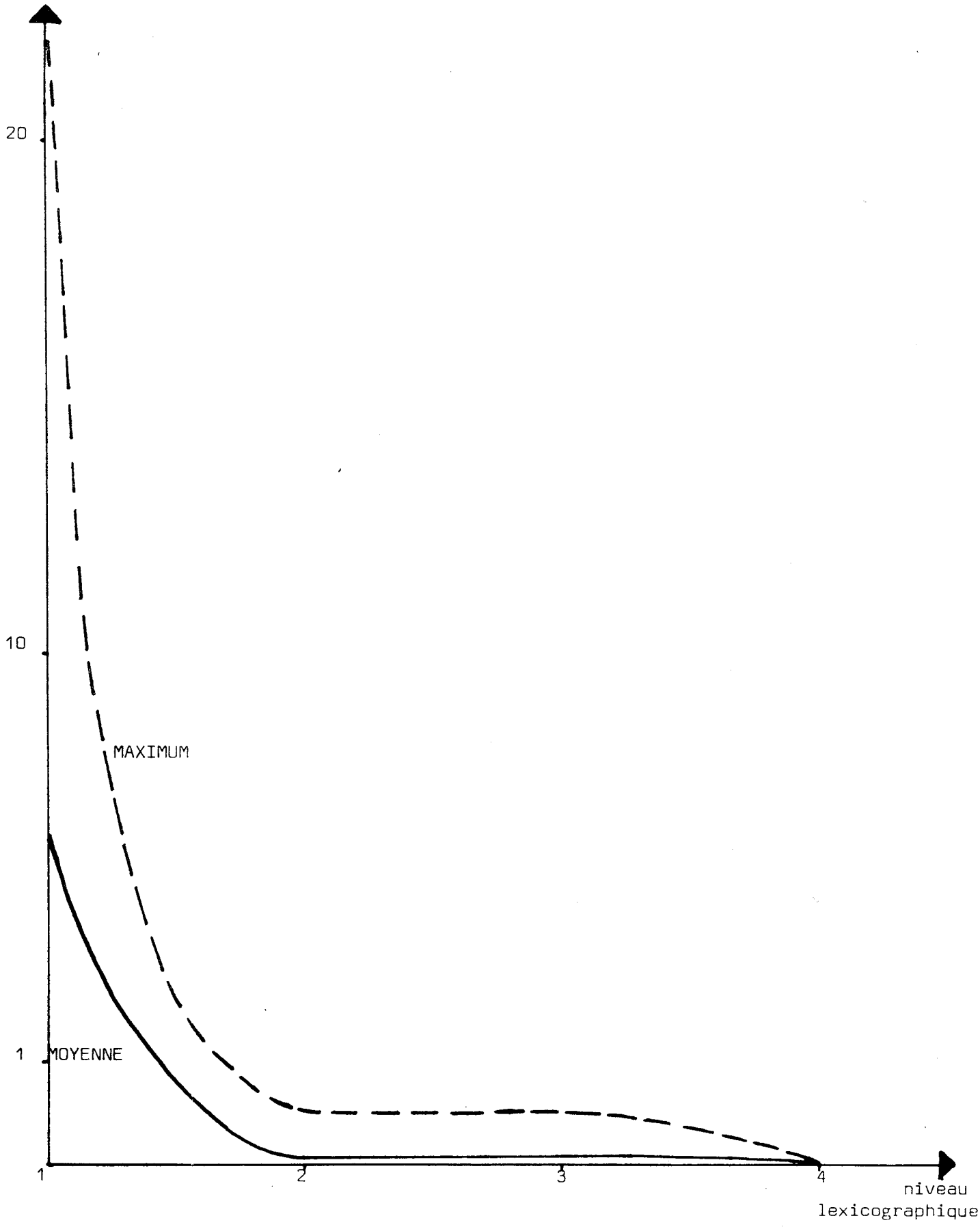


Figure III.5. Déclarations de constantes

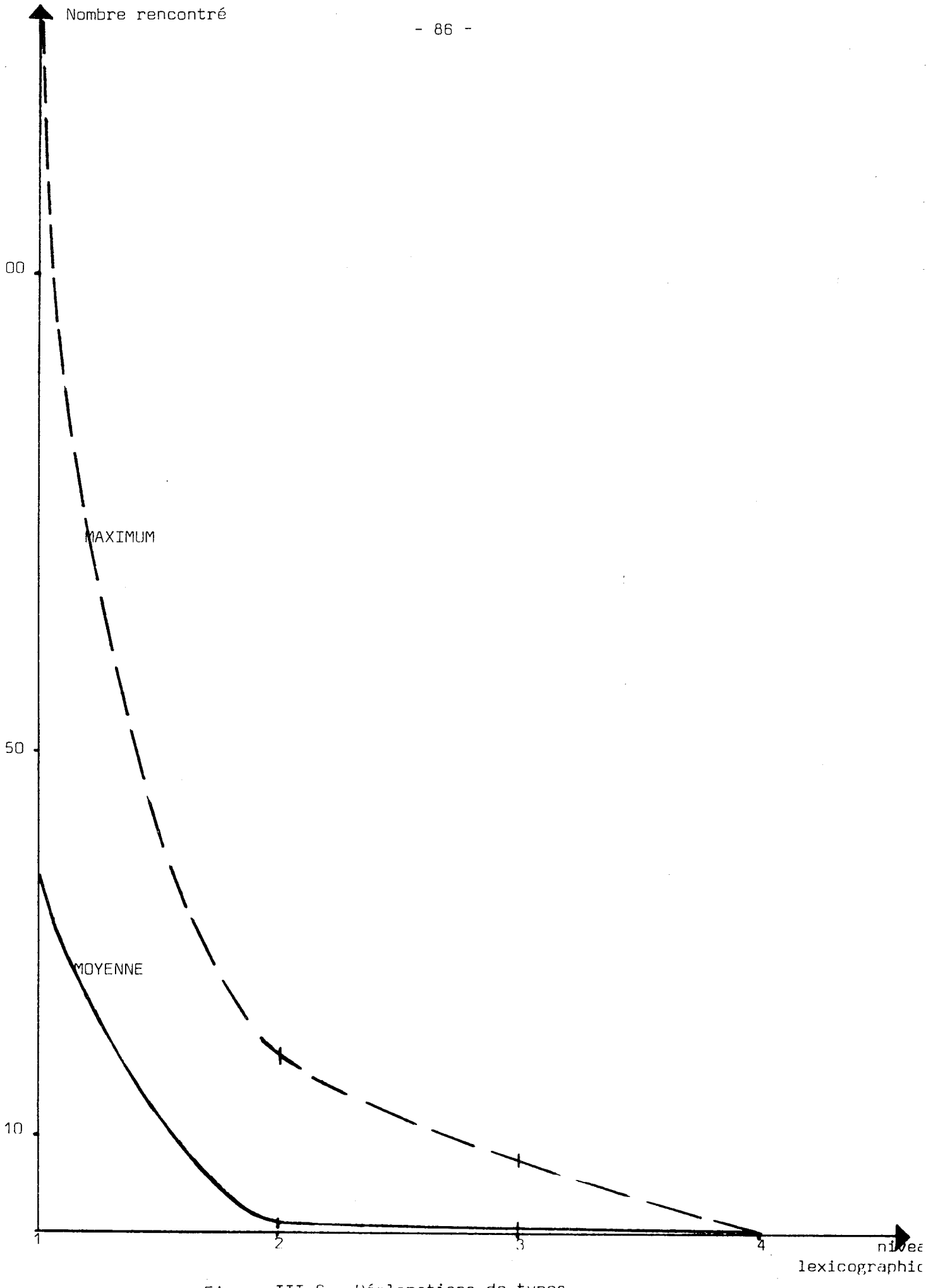


Figure III.6. Déclarations de types

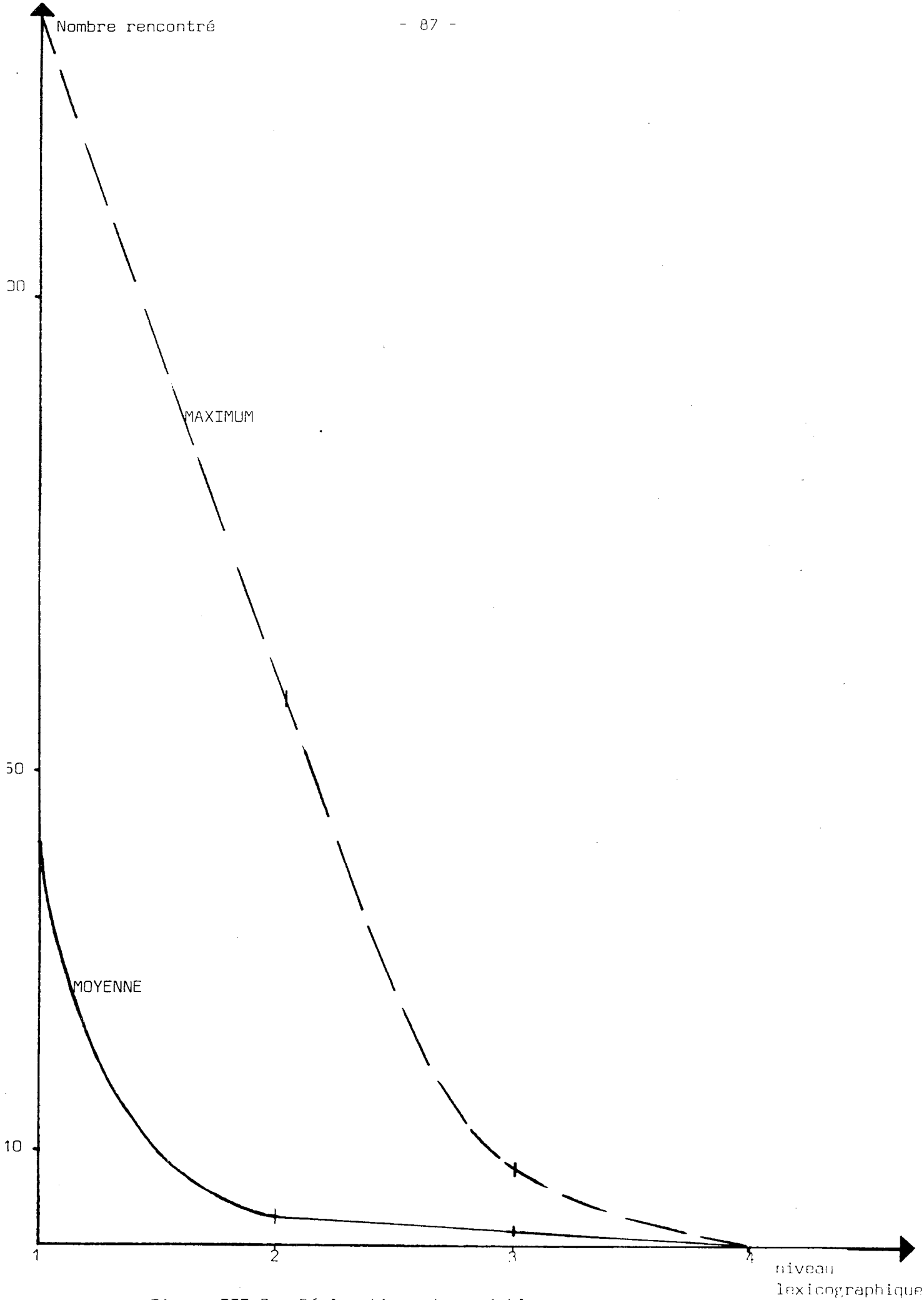


Figure III.7. Déclarations de variables

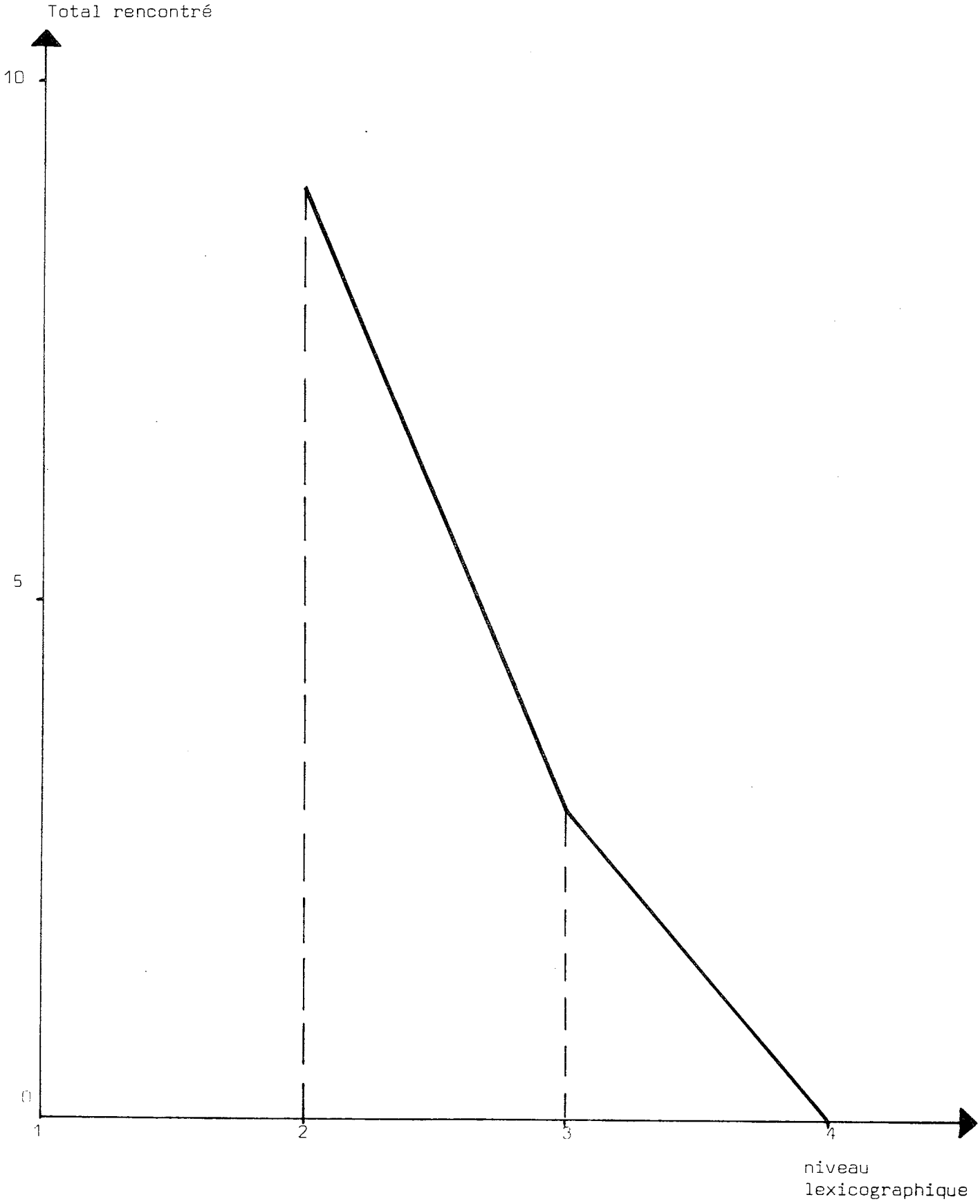


Figure III.8. Nombre maximum de paramètres

IV - LANGAGE INTERMEDIAIRE

Nous nous plaçons à priori, dans l'optique d'une machine à octet pour les formats des descripteurs et des instructions. L'adressage au niveau du bit nous paraissant trop prohibitif et des mots plus longs ne seraient pas suffisants pour condenser l'information.

La terminologie employée est celle d'IBM (mot: 32 bits, demi-mot: 16 bits).

IV.1. DECLARATION

IV.1.a) Constantes

La plupart des constantes sont déclarées au niveau global (97,6 %), nous aurons donc peu d'opération de création de constante.

D'autre part, il est possible que la référence à une constante (descripteur) prenne autant de place que la valeur de la constante elle-même.

Dans ces conditions, si la constante est suffisamment petite, il n'y a pas de création, chaque référence à cette constante étant remplacée par le littéral correspondant. Si elle est trop grande (taille supérieure à un demi-mot (voir codage des instructions)), il y a création d'un descripteur, initialisé par un littéral, l'accès étant le même que pour les variables.

IV.1.b) Variables

Il nous reste ici à définir plus complètement les déclarations de variables.

Trois considérations vont nous le permettre:

1/ les types prédéclarés sont très employés (§ II),

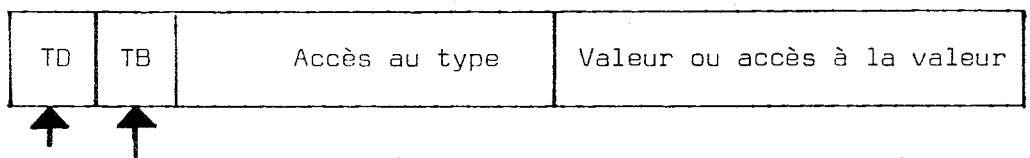
2/ nous avons besoin d'une information complémentaire pour les opérateurs (§ I.3.b),

3/ il faut minimiser le nombre d'accès mémoire.

Les considérations 1 et 3 nous amènent à mettre, dans la mesure du possible, la valeur dans le descripteur pour les types prédéclarés.

Les considérations 2 et 3 impliquent un préfixe au descripteur qui évitera d'aller lire les descripteurs de type à chaque opération.

Nous aurons finalement plusieurs sortes de descripteurs de variables, un préfixe indiquant la catégorie du descripteur est donc indispensable.



Dans le cas des entiers et réels, nous considérons que les types prédéclarés "integer" et "real" ont les limitations de la machine (type implicite).

C'est-à-dire, le type "integer" est un type intervalle d'entier dont les bornes sont les valeurs minimum et maximum pouvant être représentée dans un mot de la machine (de même pour le type "real").

D'où trois formats de descripteurs de variables:

- préfixe - accès au type - valeur(scalaires, 'char', ...)
- préfixe - valeur (entier, réel standard)
- préfixe - accès au type - accès à la valeur (structures).

Le préfixe indiquant le format (TD) et comportant la spécification à l'opérateur (TB): opération permise ou non,
type opération (-entier ou différence d'ensemble par exemple).

Dans le cas d'éléments de structure le préfixe pourra être mis en facteur. Ceci pour deux raisons:

- gain de place,
- forçage à un type.

La deuxième raison est la plus importante. L'accès à un champ qui n'a pas d'existence logique est permis en PASCAL. On peut considérer par exemple un 'powerset' comme un entier ou inversement. Dans ce cas, un OU peut être effectué sur un entier puisque le champ logique est un 'powerset'.

Ceci ne peut être fait qu'en imposant le préfixe logique comme préfixe physique.

Nous aurons besoin aussi d'une information d'indirection dans le préfixe pour les paramètres par référence.

Remarque pour les valeurs:

Avant la première affectation, la valeur d'une variable est indéfinie. Il nous parait important , pour des raisons de sécurité, de représenter explicitement cette valeur particulière. Ceci est réalisé par la présence d'un'bit d'initialisation' lié à chaque valeur.

IV.1.c) Types

Le préfixe des descripteurs de type n 'indique en général que la catégorie à laquelle appartient ce descripteur. Nous pourrons avoir en plus pour les structures (champ, tableau...) une information sur les préfixes obligatoires.

La réalisation fine des descripteurs (annexe III) nous servant de base pour les calculs, nous pouvons maintenant estimer la taille de chaque descripteur. D'où nous déduirons la taille moyenne d'un descripteur de type, d'après les mesures du chapitre II.

Taille moyenne totale d'un descripteur de Record ('nombre' est tiré du § III.3):

l'unité est l'octet

descripteur	nombre	taille du descripteur	taille totale
de record	1	4	4
de champ	5,77	6	34,62
de branche	2,7x0,44	6	7,2
total			45,8

Nous n'avons pas tenu compte des CASES emboîtés (très peu nombreux). La taille est donc de 46 octets.

D'après les mesures de pourcentage relatif des types (III.3.), nous obtenons:

type	taille du descripteur	pourcentage	taille dans 100 types
pointeur	2	3,8	7,6
tableau, classe	4	37,1	148,4
pseudo-vecteur...	4	45	180
fichier	6	1,3	7,8
sous réel, sous entier	8	2,2	17,6
record	46	10,6	486,8
total			848,2

La taille moyenne d'un descripteur de type est donc de 8,48 octets.

IV.1.d) Procédures et fonctions

Ils sont conservés tels qu'ils sont définis au chapitre I.

Les renseignements concernant les déclarations locales et les paramètres sont limités à leur nombre. Les descripteurs de paramètres suivent immédiatement le descripteur de procédure.

IV.1.e) Noms internes

Calculés par référence à la réalisation fine (annexe III).

Les descripteurs de variables sont cadrés sur un mot (4 octets) ; ceux de procédure vont utiliser deux mots.

Nous pourrons maintenant estimer la valeur maximum susceptible d'être atteinte par le numéro de déclaration, les maxima des différentes déclarations étant connus (chapitre III).

Pour un programme complet, les maxima cumulés des différents niveaux (compilateurs):

- 175 variables déclarées,
- 44 procédures et fonctions,
- ∿80 paramètres.

Les descripteurs de procédure sont sur deux mots.

Nous obtenons 343 mots au total d'où un numéro de déclaration sur 9 bits. De plus, 3 bits sont pris pour le numéro de niveau dans le cas intermédiaire, ce qui laisse 6 bits dans ce cas là.

Il y a au maximum 62 déclarations au niveau 2 (paramètre compris).

Nous prendons donc, par sécurité, un numéro de déclaration de 10 bits. En appliquant le même raisonnement pour les types, nous otenons 153 types au maximum de taille moyenne 8,5 octets, d'où environ 1300 octets.

Les descripteurs de type étant des multiples de demi-mot (2 octets), 10 bits de nom interne (adressage par demi-mot) sont suffisants.

Comme pour les variables, nous prendrons un bit supplémentaire par précaution, d'où un nom interne de 11 bits.

IV.2. INSTRUCTIONS

Nous avons vu au § II que nous faisons des contrôles sur les instructions. Ce contrôle se fera par la création d'un descripteur de contrôle. Nous appellerons de tels descripteurs,des descripteurs de segment (terminologie de WORTMAN).

Un segment est l'ensemble des instructions appartenant à une instruction composée (corps). Le nom interne d'un segment est son adresse de début (ou de fin), c'est l'information principale contenue dans le descripteur de segment.

Les segments définis ici sont moins fins que ceux utilisés par WORTMAN (3 segments pour le groupe "do while" de PL, un seul ici) ou que ceux utilisés dans le compilateur incrémentiel d'ALGOL [GR-70].

Les instructions de contrôle du langage intermédiaire seront essentiellement des instructions de manipulation de tels descripteurs.

Comme dans les déclarations, nous allons maintenant simplifier et minimiser les instructions d'accès ; les autres instructions seront conservées avec les modifications apportées au chapitre III.

IV.2.a) Choix

L'évaluation de l'expression est indépendante de l'environnement de contrôle des instructions de choix et peut être effectuée avant l'entrée dans celui-ci.

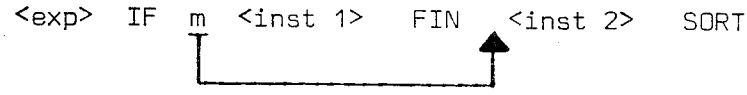
De plus, en plaçant l'entrée de l'environnement après l'évaluation de

de l'expression on peut condenser deux instructions en une seule (gain de place) et donc effectuer le test avant la création du descripteur. Celui-ci est construit seulement si le test est vrai (pour 'if') d'où un gain de temps d'exécution.

La fréquence statique importante de l'instruction 'if' (résultat corrobor par la fréquence dynamique, § VI.3) rend ces modifications très rentables

On peut généraliser ce résultat aux instructions conditionnelles. Mais ce n'est pas le cas pour les instructions répétitives (réévaluation de l'expression à chaque répétition).

- instruction 'if':



CASE

Dans une première phase, nous choisirons la structure de test séquentiel. Cette instruction étant peu employée, l'instruction CASE sera mise aussi après l'expression.

Il est intéressant de spécifier dans le CASE, la taille de l'instruction. Nous calculerons alors l'adresse de fin qui figurera dans le descripteur de Segment.

Les instructions de Fin d'alternative n'ont plus alors aucun paramètre. L'adresse FIN de l'instruction 'case' est spécifiée une seule fois.

Le Nom interne des segments conditionnels est l'adresse fin de segment. L'instruction 'FIN' utilise cette adresse pour la sortie de l'environnement.

IV.2.b) Boucle

Le 'while' et le 'Repeat' indiquent tous deux l'entrée dans une boucle simple. Nous regroupons ces deux instructions en une seule instruction: 'BOUCLE', ce qui a l'avantage de simplifier les segments (moins de segments différents).

La particularisation de l'instruction de fin de segment ('SORT') pour l'instruction PASCAL 'Repeat' est résolue en contractant le couple d'instructions 'SORT', 'UNTIL' en une seule instruction 'UNTIL' qui effectuera aussi maintenant la sortie de l'environnement dans le cas "expression vraie".

De même, nous pouvons particulariser l'instruction de fin de boucle 'while': instruction 'CYCLE'.

Maintenant l'instruction 'SORT' n'apparaît plus que dans la traduction des instructions PASCAL de choix, d'où une interprétation simplifiée.

Formes définitives des boucles PASCAL:

```
BOUCLE                    BOUCLE
  <instlist>                <exp>
  <exp>                   DO      m
UNTIL                       <inst>
                          CYCLE
```

En ce qui concerne l'instruction 'For' nous avons plusieurs remarques:

- les expressions sont évaluées en une seule fois à l'entrée comme pour les instructions conditionnelles, d'où un déplacement possible du 'For up' ('down') après ces expressions.

- la variable de contrôle pour la boucle est une variable simple.

Nous pouvons donc utiliser l'affectation prévue pour ce cas (§ III.7.b).

De plus, dans la mesure où 'FORUP' est placé après les expressions, il est susceptible de faire le premier test, d'où maintenant un post-fixage de l'opérateur 'TO' est possible.

Nous otenons alors:

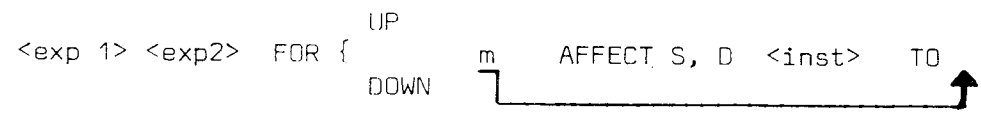
	|<exp 1>	
	|<exp 2>	
	FORUP (DOWN)	test entrée éventuelle dans l'environnement
	AFFECT S_{var}, D_{var}	
	|<inst>	
segment	TO (DOWN)	incrément(décrément), test et bouclage éventuel
Forup (down)	SORT	sortie du segment

Remarques:

Comme pour les autres boucles de PASCAL, en particulier le 'repeat', nous pouvons regrouper les deux dernières instructions d'où une économie de place.

La variable de contrôle pour la boucle est une variable déclarée du programme, son incrémentation par 'TO' impose que celui-ci ait l'adresse de cette variable.

Le descripteur précisant la croissance ou décroissance, une seule version de l'instruction est nécessaire ('DOWN' inutile).



Ceci impose que le descripteur de segment'For' contienne outre son adresse début (nom interne), la valeur de la limite.

IV.2.c) With

Les variables "déclarées" dans l'instruction "with" peuvent être, comme pour le 'IF' ou le 'CASE', calculées une fois pour toutes avant l'entrée

dans l'environnement de cette instruction "with".

D'où le déplacement du "WITH" comme pour les instructions conditionnelles.

WITH		
<varlist>	|<varlist> }	calcul des "paramètres"
do	WITH	'empilage' des n descripteurs calculés création du descripteur de segment with
<instlist>	|<inst>	
	SORT	

Les variables spécifiées dans le 'with' ne sont pas des déclarations ni des paramètres. Leur portée est limitée statiquement au 'with'.

Exemple:

Toute procédure appelée à l'intérieur d'un 'with' ne peut avoir comme paramètre, ni référencer dans son corps, les variables "déclarées" dans le 'with' [WI-74] (à moins de les recalculer).

On ne peut donc pas considérer ceci comme un bloc ; ce sera un segment particulier.

IV.2.d) Fin de segment

L'instruction de fin de segment est l'instruction 'SORT' qui "dépile" le descripteur du segment courant (correspond à ';').

L'instruction 'FIN' est une fin de segment accompagnée d'un branchement.

L'instruction 'SORT', dans la traduction du 'with' de PASCAL, dépile, outre le descripteur du segment, les descripteurs des pseudo-déclarations.

IV.2.e) Conclusion

D'après la taille du nom interne calculé précédemment (12 bits), nous avons déterminé un format fin du codage des instructions (annexe III).

Pour ceci, nous avons estimé un déplacement maximum pour les branchements sur 12 bits également. Le codage étant effectué suivant la fréquence, avec la contrainte d'avoir un nombre entier d'octets, grâce aux mesures statiques, nous avons calculé grossièrement la longueur moyenne d'une instruction, suivant le même principe que pour les types.

Cette longueur est d'environ 16 bits (2 octets).

La longueur maximum du code d'une procédure accessible par de tels branchements est de ±2048 instructions.

A titre de comparaison, la plus grande procédure rencontrée (qui appartient au compilateur) a une taille estimée à moins de 1500 instructions de langage intermédiaire.

Le déplacement choisi sur 12 bits, nous parait donc suffisant.

Remarque:

Pour le compilateur, le facteur Taille code 360/(Taille code L.I. + descripteurs de type) est voisin de 4(120K octets/30K octets).

Avec la pile de contexte, le facteur est voisin de 2.

IV.3. TRADUCTEUR

IV.3.a) Spécifications

Le traducteur génère le langage intermédiaire décrit précédemment en un seul passage.

Plusieurs possibilités supplémentaires ont été introduites:

- les opérandes de l'affectation sont réordonnés:

  <variable> := <exp> devient <exp> <variable> :=

- les déclarations et le corps d'une procédure sont concaténés. Nous aurons donc en tête du programme objet, les procédures de plus haut niveau. Le programme principal est à la fin.

- le 'end' du programme est traduit en une instruction 'STOP' (la fin de bloc normal est l'instruction 'Retour').

- les adresses relatives (taille du segment) devant être calculées pour les instructions composées, le contrôle sur les 'goto' simples est effectué. L'instruction GO aura donc un deuxième paramètre qui est le nombre du segment dont on sort par le 'goto'.

- les contrôles d'erreur sont limités et il n'y a pas de procédure de récupération. Une erreur sémantique provoque donc l'arrêt de la traduction.

Le traducteur n'est qu'un simple outil, pour obtenir automatiquement du langage intermédiaire servant à tester l'interpréteur.

Une version complète sera écrite ultérieurement en PASCAL, et par génération télescopique ("Bootstrap") un vrai traducteur pourra être obtenu.

- le symbole "Packed" est ignoré puisque l'information est déjà condensée dans le langage intermédiaire.

Remarque:

Une version plus fine pourrait utiliser ce symbole, "packed 0..7" indiquerait alors que la valeur est sur trois bits, cette version imposant alors un adressage au niveau du bit.

IV.3.b) Structure fonctionnelle

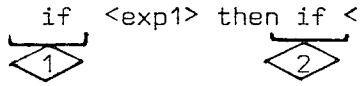
La structure du traducteur est la même que celle du Suiveur Syntaxique (annexe I).

Pratiquement il est divisé en trois modules.

. module syntaxe: interpréteur et analyseur lexicographique,
. module sémantique: algorithme de traitement des déclarations et génération des descripteurs.
. module de macros: traitement des étiquettes et génération de code.

La structure de ce traducteur est représentée par la grammaire PASCAL avec les appels de fonctions, donnée en annexe.

Il existe une ambigüité dans le langage PASCAL, celle des 'if' imbriqués:



Pour nous le else s'applique au if numéro 2. Ceci est réalisé par la fonction 'FIGE' qui empêche l'Analyseur lexicographique d'avancer à l'appel suivant.

En effet, la version utilisée du "module de syntaxe" applique le 'else' au dernier 'if' rencontré.

Nous sommes alors dans la règle: (grammaire mise sous forme LL(1))

<else stat> ::= { 'else' <statement>
                { ( )

Le caractère syntaxique rencontré après <inst2> permet de savoir dans ce cas que nous sommes dans l'alternative vide.

Puis, normalement l'analyseur lexicographique avance d'un caractère syntaxique. Or, le caractère qui nous a servi à déterminer l'alternative vide est en fait un caractère appartenant à une autre règle (celle à laquelle appartiennent les deux 'if' imbriqués) et il est perdu.

La fonction 'FIGE' sauvegarde donc ce carcatère pour la suite de l'analyse, une fois le choix de l'alternative fait.

Remarque:

Dans la version PASCAL, pour la portabilité du langage [WI-72] une instruction 'Loop <instlist> exit if <exp> <instlist> end' a été ajoutée. Le traducteur accepte cette instruction, mais celle-ci n'ayant pas été maintenue dans la dernière version du langage, n'a pas été traitée ici.

V.- MACHINE ABSTRAITE

V.1. INTRODUCTION

L'interprétation va s'articuler autour de trois piles formelles et deux tables:

- table de type construite par le traducteur,
- table du code construite par le traducteur,
- pile des descripteurs, initialisée par les descripteurs globaux et prédéfinis,
- pile des valeurs (valeurs des éléments de structure),
- pile de contrôle.

L'adressage des variables se fera dans la pile des descripteurs par l'intermédiaire d'au moins deux variables: 'Global' (base de la pile) et 'local' (variables locales).

L'adressage spécial dans l'instruction 'With' se fera par l'intermédiaire du descripteur de contrôle du 'With' repéré dans la pile par une variable 'segment'. De plus, une variable de gestion (zône libre) est nécessaire pour chaque pile.

Remarque:

Ces trois piles pourront être réduites à une dans la phase suivante. Il n'y aura plus alors qu'une seule variable de gestion: 'LIBRE'.

Des variables de base pour les deux tables sont aussi indispensables: 'Basetype', 'Base code' et aussi l'adresse courante dans le code: 'co'.

Toutes les variables introduites à ce niveau sont nécessaires (mais non suffisantes) à la gestion de la mémoire.

Au niveau de l'unité centrale, nous disposerons de plusieurs variables pour le décodage et l'utilisation des instructions et des descripteurs, en plus du mécanisme d'évaluation.

Organisation

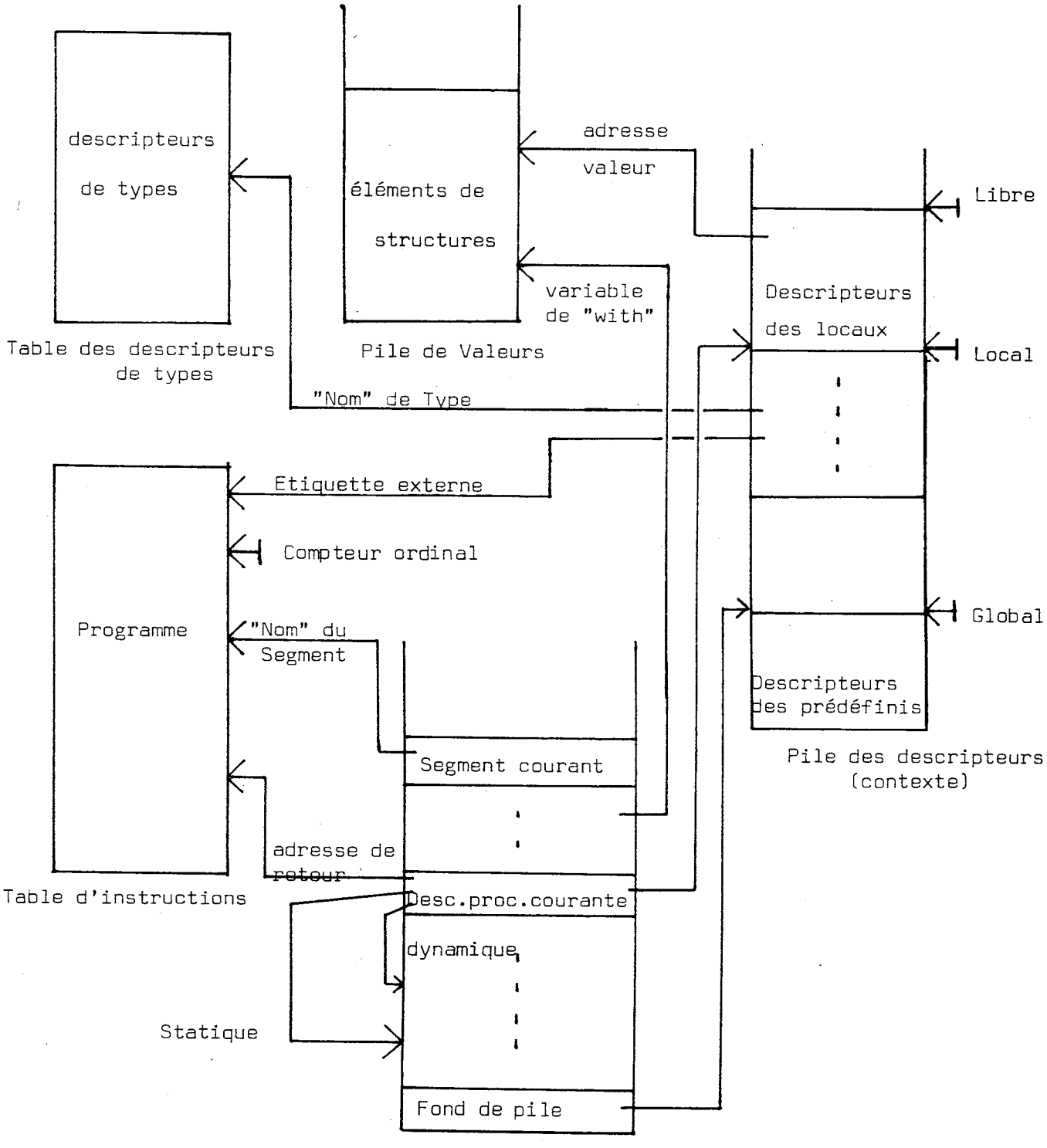


Figure V.1. Organisation

V.2. INSTRUCTIONS D'ACCES

NOM S,D : calcule l'adresse et lit le descripteur de la variable de nom interne "S,D".
Si la variable est une variable simple, nous disposons alors de sa valeur, sinon nous disposons de l'adresse de la valeur (l'indirection, si elle existe, est exécutée)

CHAMP p : calcule l'adresse et lit le descripteur de champ numéro 'p' dans la variable "record" précisée précédemment.
(le contrôle d'existence est effectué)

IDENT S,D : calcule l'adresse et lit le descripteur de la variable de nom interne "S,D". Cette instruction construit, de plus, un descripteur d'indirection sur cette variable.
(cette instruction est utilisée pour le passage d'un paramètre par référence, dans le cas d'une variable simple).

POINT n : modifie le descripteur de pointeur ou de fichier venant d'être lu en un descripteur d'indirection.
L'indirection est exécutée 'n' fois.
(dans le cas d'un fichier n=0)

INDEX (S,D;)n :
exécute n indexations sur le tableau spécifié. Le résultat est un descripteur d'indirection vers l'élément du tableau sélectionné.

L'indexation s'effectue par un calcul à partir des éléments du pseudo-vecteur. L'élément de pseudo-vecteur, pour une dimension, est composé de:

DV1: 1er valeur de l'élément de pseudo-vecteur,
DV2: 2° valeur de l'élement de pseudo-vecteur.

Avec cette convention d'écriture le calcul de l'adresse de l'élément de tableau est donné par la formule:

$$\text{adresse élément} := \text{adresse racine du tableau} + \sum_{k=1}^{n} (\text{Valindex}_k - DV1_k) * DV2_k$$

Le contrôle de dépassement des bornes pour chaque dimension est:

$$Valindex_k - DV_{1_k} > 0 \quad \text{sinon erreur}$$

$$(Valindex_k - DV_{1_k}) * DV_{2_k} < DV_{2_{k-1}} \quad \text{sinon erreur}$$

CHAMP i,j;p : lit la jème variable du i^e "with" (récursivité) puis exécute une instruction 'CHAM p'.

AFFECT (S,D) : calcule l'adresse et lit le descripteur de la variable de nom interne S,D. Puis vérifie que la valeur à affecter est compatible avec le type de la variable. Dans l'affirmative, l'affectation proprement dite est exécutée (sinon il y a erreur).

V.3. LITTERAUX

UNZER P,Φ : construit dans la pile d'évaluation un descripteur de valeur avec le préfixe P et la valeurΦ(0 ou 1).

LITT P,V : même chose que pour l'instruction précédente, la valeur V étant ici une valeur sur 8 bits (scalaire, chaîne) ou 16 bits (entier court) ou 32 bits.

Dans le cas où P indique une chaîne de caractères, V désigne sa longueur et le premier élément de la chaîne est à l'adresse suivant l'instruction 'LITT'.

Dans ce cas, celle-ci construit un descripteur de'littéral chaîne' comportant la longueur de la chaîne, dans la pile d'évaluation.

Remarque:
Une extension à des littéraux entiers d'une taille quelconque peut être faite par ce mécanisme (entier de 250 bits par exemple).

BIT n : crée un littéral 'powerset' comportant les n élements calculés précédemment. Le codage se fait grâce au descripteur du type de base. Si n = 0 création du littéral 'powerset' vide (c'est-à-dire ne comportant que des 0).

V.4. INSTRUCTIONS SPECIALES

α fonctions standards considérées comme des opérateurs.

SUCC : incrémente l'opérande de 1.

PRED : décrémente l'opérande de 1.

ODD : donne la parité de l'opérande (résultat booléen).

INT : convertit le caractère en entier.

EOF, SD : teste si le fichier de nom interne S,D est terminé (en lecture). Le résultat est un booléen.

β procédures standards

RESET S,D : dépile dans la variable classe définie précédemment jusqu'à l'adresse contenue dans le pointeur 'S,D'.

ALLOC S,D;n: allocation dans la variable 'classe' définie précédemment d'un nouvel élément dont la taille est déterminée par calcul à partir des n expressions précédentes. L'adresse de cet élément est affectée au pointeur 'S,D'.

Si n=0 la taille prise est celle qui figure dans le descripteur de type de l'élement de la 'classe'.

Si la variable 'classe' est pleine, la valeur affectée au pointeur est 'NIL'.

V.5. INSTRUCTIONS POUR LES PROCEDURES ET FONCTIONS

APPEL S,D;n: calcule l'adresse et lit le descripteur de la procédure (ou fonction) de nom interne S,D, met à jour le contexte de contrôle (marquage de la pile) ; à ce moment, une sous-instruction PARAM se charge de la vérification des types des 'n' paramètres effectifs et du passage de ces paramètres grâce aux descripteurs de paramètres formels. Enfin l'entrée dans le nouveau contexte d'adressage et le branchement au début de la procédure a lieu.

n=0 indique que c'est une procédure sans paramètre.

RETOUR : provoque le retour de la procédure avec dépilage et restauration du contexte d'appel.
Si c'est une fonction, la valeur de cette fonction est mise dans la pile d'évaluation.

Remarque:

Si le descripteur de contrôle de la procédure est de niveau 1 (global), c'est la fin du programme.

On peut supposer que ceci indique un retour dans le bloc du système.

V.6. DECLARATIONS

Les déclarations se traduisent ici par des instructions qui vont empiler les descripteurs de variables déclarées dans la pile de contexte.

CREER n,P : empile n descripteurs de préfixe P pour les variables de type standard (entier, réel, booléen, char...).

CREER n,DV : empile n descripteurs 'DV' ; accès éventuel au descripteur de type si la taille n'est pas indiquée par le préfixe (structures) et réservation de la place pour la valeur des éléments (DV: préfixe et nom interne du type des variables).

CREER 1 : empile les 1 descripteurs de procédure (et paramètre).

Remarque:

La réservation de place pour les valeurs s'accompagne de la mise à "non initialisé" du bit d'initialisation.

V.7. INSTRUCTIONS DE CONTROLE

En règle générale, elles sont générées aussitôt le terminal correspondant rencontré.

Le descripteur de segment utilisé dans ces instructions est le sommet de la pile de contrôle.

V.7.a) Instructions répétitives: 'while' et 'repeat'

BOUCLE : crée sur la pile de contrôle un descripteur de boucle comportant l'adresse de début du segment 'boucle'. Cette adresse sert au bouclage et au contrôle des branchements éventuels à l'intérieur de la boucle par un 'goto'.

DO m : teste le résultat de l'expression calculée précédemment:
- . vrai: non opération
- . faux: branchement à adresse courante+m et dépilage du descripteur de segment 'bouclé'.

CYCLE : lit le descripteur de boucle et provoque un branchement à l'adresse contenue dans ce descripteur.

UNTIL : teste le résultat de l'expression évaluée précédemment:
- . vrai: dépilage du descripteur de segment 'boucle',
- . faux: branchement à l'adresse contenue dans le descripteur de segment 'boucle'.

Instructions FOR

FOR/UP DOWN : duplique la valeur finale, puis vérifie que la valeur finale est supérieure (inférieure) à la valeur initiale. Si ce n'est pas le cas alors il y a branchement à l'adresse courante+m (sortie de la boucle). Si le test est vérifié, alors il y a création d'un descripteur de segment 'FORUP'(DOWN) comportant la valeur de la limite et l'adresse de début du segment.

TO S,D : lit le descripteur de segment 'FOR', et, en fonction du type de boucle FOR (croissante ou décroissante), incrémente (décrémente) la valeur de la variable de nom interne S,D, et la compare à la valeur de la limite.

Si elle est inférieure (supérieure) à la limite, il y a branchement à l'adresse contenue dans le descripteur de segment, sinon il y a sortie de l'environnement par dépilage de ce descripteur.

V.7.b) Conditionnelles

Instructions IF

IF m : teste - si vrai: construction d'un descripteur de IF avec l'adresse de fin du segment (nom interne),
si faux: branchement à adresse courante+m.

IFELSE $m_1 m_2$: construit un descripteur de segment IFELSE avec adresse de fin de segment (adresse courante+m_1), puis teste:
- . vrai: passage normal en séquence,
- . faux: branchement à adresse courante+m_2.

SORT : marque la fin de l'instruction de contrôle: dépilage du descripteur.

Instructions CASE

CASE m : construit un descripteur de segment 'choix' avec l'adresse de fin de segment.

OF n,{e_i}*,m:

teste si la valeur de l'expression est dans la liste des 'n' étiquettes 'e_i'.

En PASCAL: exp IN [e_1,...e_n]?

- si oui: passage normal en séquence,
- si non: branchement à adresse courante+m.

Remarque:

Si m=0 ceci indique la dernière alternative ; dans ce cas, si le test est faux, il y a déroutement vers une procédure d'erreur.

FIN : accède le descripteur de segment 'choix', provoque le branchement à l'adresse contenue dans ce descripteur et le dépilage de ce dernier.

V.7.c) Instruction with

WITH n : empile un descripteur de segment 'with' et les n variables de type record calculées précédemment.
Cette instruction initialise aussi le mécanisme spécial d'adressage pour les instructions CHAMW (mise à jour de SEG)

Remarque:

Le dépilage (fin d'instruction 'with') se fait par l'instruction 'SORT'.

V.8. LES BRANCHEMENTS

GO n, ns : provoque le branchement à l'adresse 'n' et le dépilage de 'ns' descripteurs de segment.

EXIT S,D : dépile dans la pile de contrôle jusqu'au niveau S, puis accède le descripteur d'étiquette externe et dépile les descripteurs de contrôle jusqu'à retrouver le même nom qu celui spécifié dans le descripteur de l'étiquette (si non trouvé, alors erreur) et enfin provoque le branchement à l'adresse contenue dans le descripteur d'étiquette.

V.9. OPERATEURS

Tous les opérateurs accèdent la valeur du ou des opérandes s'ils ne so pas déjà dans le(s) descripteur(s) de variable. Le préfixe de la vale indique si l'opération est permise et précise l'opération. Elle est alors exécutée. Le résultat est mis à la place de l'opérande supérieu dans la file (ou pile).

opérateurs disponibles: EGAL, DIFF, SUP, INF, SUPEG, INFEG, IN
PLUS, MOIN, OU
MULT, DIVI, DIV, MOD, ET
NON, NEG.

VI.- MESURES DYNAMIQUES

VI.1. INTRODUCTION

Le résultat de ces mesures sera essentiellement utilisé pour le passage de l'intepréteur actuel à l'interpréteur de la "micro-machine". Les instructions les plus fréquentes doivent être optimisées. Ceci sera particulièrement important pour le découpage des algorithmes en "primitives".

Le choix de ces primitives et de l'implantation des variables sera fait en fonction du résultat de ces mesures et de contraintes extérieures comme les propriétés de la micro-programmation et la puissance globale désirée pour la future machine.

Une autre utilisation de ces mesures consiste à modifier légèrement le pas précédent. Ici, il s'agit du langage intermédiaire lui-même (bouclage dans la démarche).

Ces modifications sont en principe très limitées.

Enfin, ces mesures peuvent influencer la manière même d'écrire l'interpréteur. Cette influence est très floue et nous ne pouvons pas donner de règles précises.

VI. 1.a) Programmes mesurés

Les programmes PASCAL utilisés pour ces mesures proviennent pour la moitié de l'Université d'AARHUS et pour l'autre moitié de programme réécrit en PASCAL à l'Université de GRENOBLE.

L'éventail des tailles est plus restreint que pour les mesures statiques, le plus gros programme a 907 lignes, le plus petit 51.

Les écarts (en pourcentage) enregistrés entre deux exécutions d'un même programme étant très importants, le nombre de lignes a, en fait, peu d'importance.

Programmes d'AARHUS:

- Arborescence binaire: lit une expression parenthésée et construit les formes postfixée, préfixée et infixée.
- Nombre aléatoire: calcule une série de nombres "aléatoires" dans un tableau.
- Références croisées.
- Occurences de mots clés.

Programmes obtenus en transformant en PASCAL des programmes écrits en ALGOL W par des étudiants en analyse numérique:

- Résolution d'un système linéaire par la méthode de GAUSS (triangularisation de matrice).
- Intégration par la méthode de SIMPSON.
- Tri sur le tas ("Heap-Sort").

Enfin un programme écrit par des étudiants de D.E.A. qui est un transformateur de grammaire simplifié:

- Il vérifie les quatre conditions de KNUTH pour qu'une grammaire soit LL(1) et génère des appels de macros.

VI.1.b) Remarques sur les mesures

Le nombre d'instructions comptées est de 60 261 593.

Les mesures provenant d'erreurs dans les programmes (bouclage notamment) n'ont pas été comptabilisés dans ce nombre.

Etant donné les différences très importantes dans la taille des programmes, nous n'avons pas additionné les occurences de chaque instruction, mais fait un pourcentage pour chacune des huit catégories de programmes.

L'avantage de cette méthode est qu'elle nous donne un pourcentage plus proche de la réalité et nous permet d'avoir en plus un écart-type.

Cet écart-type est très important pour des instructions peu fréquentes (Reset, abs, ...), il est même supérieur à la moyenne. Nous préciserons seulement le minimum et le maximum. Quant à celles dont la probabilité est pratiquement nulle, nous indiquerons seulement la fréquence avec leur pourcentage dans ce programme.

Dans ces mesures, il n'a pas été fait de séparation entre les instructions "NOM" et "ID", cette dernière étant très peu fréquente (variable simple passée en paramètre par référence).

Il en est de même pour les littéraux, qui ont été comptés globalement ('LITT' et 'UNZER' non séparés) à l'exception des littéraux "Powerset" qui sont particuliers.

Des mesures particulières à chacune de ces instructions et au préfixe du littéral seront nécessaires au niveau d'interprétation suivant.

D'autre part, les instructions INDEX S,D;n et POINT S,D ont été comptées en respectivement NOM S,D + INDEX n et NOM S,D + POINT, ces instructions ayant été introduites uniquement pour une compaction de code. Au niveau de l'exécution, INDEX S,D;n est pratiquement équivalente à NOM suivi de INDEX, si ce n'est pour la lecture et le décodage de l'instruction.

Dans le cas de l'affectation, nous n'avons pas séparé l'affectation à une variable simple ("AFFECT S,D") de celle à un élément de structure ("AFF").

Seule l'affectation à la variable de contrôle de la boucle (cas "AFFECT S,D") a été comptée à part. Le point intéressant à ce stade est de connaître la fréquence de l'affectation proprement dite.

Le compteur NOM comptabilise donc en fait, non pas le nombre d'instructions NOM, mais le nomre d'accès en général à un descripteur de variable à partir de son nom interne.

La fréquence dynamique obtenue est celle du calcul d'adresse du descripteur de la variable et sa lecture, ce qui est le principal du point de vue fonctionnel.

- Constatations:

Dans tous les programmes mesurés (statiquement ou dynamiquement) il n'a jamais été trouvé plusieurs indirections sur un pointeur (c'est-à-dire plus d'un symbole "↑" à la fois).

D'autre part, assez souvent (toujours pour les fichiers) l'indirection sur un pointeur est de la forme "<ident>↑".

L'instruction "POINT" est donc inutile (n toujours nul) et pourrait être remplacée par une instruction "POINT" avec éventuellement une deuxième forme "POINT S,D" comme pour l'indexation. Mais les fréquences statiques et dynamiques de cette instruction ne justifient pas cet aménagement.

VI.2. OUTIL

Les mesures ont été effectuées par l'intermédiaire du compilateur IREP-CICG, ceci offrant plusieurs avantages:

- les mesures ont pu être réalisées plus tôt,
- économie de temps de machine,
- tests supplémentaires pour ce compilateur,
- facilité d'insertions d'instructions de comptage dans le compilateur.

Un exemplaire du compilateur source a été modifié, puis compilé par le compilateur actuel ("bootstrap").

Les modifications sont les suivantes:

- adjonction d'une variable tableau d'entiers prédéclarée par le compilateur (COMPTEUR),
- adjonction d'une option supplémentaire ("Piège"),
- insertion des ordres de génération de comptage si l'option "Piège" est vraie
  (génère 3 instructions IBM:

```
L  12, (Ad. compteur)
LA 12, Increment (12)
ST 12, (Ad. compteur)
```

Nous disposons donc maintenant d'un compilateur, permettant un comptage à la demande (mise en effet de l'option de piégeage qui est initialisée à Faux).

Les fichiers prédéclarés n'existant pas encore, la sortie des valeurs des compteurs est laissée au soin du programmeur.

Remarque:

Un sous-produit immédiat de cet outil a été un compilateur générant à la demande des instructions de trace dynamique qui utilisent le même tableau d'entiers.

Exemple d'utilisation:

```
LABEL
CONST                                          déclarations normales
TYPE                                              du programme
VAR
SORT: file of array (0..132) of char ; fichier de sortie des
                                       compteurs
```

```
Procédure SORCOMPT ;      /* Procédure de sortie des compteurs
  Var K, N1, POS, N : Integer :
      D: 0..9 ;
  begin
      for K:=0 to NBCOMPT do begin
        POS:=20; N:=$COMPTEUR(K);
      SORT":= ' ';
      Repeat N1:=N DIV 10 ; D:=N - N1*10;
      N:=N1 ; SORT"(POS):=chr(D+int('0'));
      POS:=POS-1  until N-0 ;
      PUT (SORT) end ;
  end ; /* SORCOMPT */
       /* $P + */
```

conversion en caractère

mise en effet de l'option de comptage

déclaration des procédures et fonctions du programme

```
  begin
```

programme principal

```
       /*  $P - */  /* arrêt de l'opération comptage */
       SORCOMPT     /* appel de la procédure de sortie des
                       compteurs */
  end.              /* fin du programme à mesurer */
```

VI.3. RESULTATS DES MESURES

VI.3.a) Tableau général des pourcentages

instruction	pourcentage	écart type	mini	maxi
NOM	34,54	4,5	25,9	45,8
LITT	10,21	2,72	5,04	13,2
AFFECT(:=)	9,52	1,42	7,73	11,38
INDEX	6,72	3,01	0,61	13,11
PLUS(+)	3,38	0,79	2,1	5,8
IF	2,6	1,34	4,63	0,05
APPEL	2,27	0,98	0,01	6,2
RETOUR	2,27	0,98	0,01	6,2
SORT	1,95	0,78	0,06	3,03
POINT(↑)	1,77	1,37	0,02	7,25
IFELSE	1,76	1,24	0,04	3,21
SUP(>)	1,75	0,57	0,79	2,34
CREER	1,53	0,96	0	3,2
EGAL(=)	1,52	1,09	0	3,5
ET(&)	1,45	0,89	0	3,01
DO	1,40	1,1	0	4,3
CHAMP(.)	1,4	1,2	0	3,5
MOIN(-)	1,4		0	6,4
CYCLE	1,33	0,9	0	2,21
INFEG(≤)	1,32		0	5,1
TO	1,25	1	0	3,38
MULT(*)	1,2		0	5,1
SUPEG(≥)	1,05	0,95	0	3,03
FIN	0,91	0,57	0,01	2,21
DIFFER(≠)	0,84	0,77	0	2,86
UNTIL	0,74	0,55	0	1,42
BOUCLE	0,52	0,37	0,01	1,67
INF(<)	0,45		0	1,02
OF	0,44		0	2,34
Get+Put+Eof	0,35		0	0,81
GOTO	0,33		0	1,4
DIV(/)	0,31	seulement	0	2,21
CASE	0,27	pour calculs	0	1,17
CHAMW	0,25		0	0,8
FORUP	0,22		0	1,05
DIVENT	0,19		0	0,49
OU.(|)	0,12		0	0,79
WITH	0,11		0	0,42
IN	0,09		0	0,7
MOD	0,07		0	0,4
ALLOC	0,06		0	0,39

En ce qui concerne les autres instructions, les pourcentages sont trop faibles pour être significatifs:

ABS(0,02%), NON(0,02%), FORDOWN(0,01%), BIT,STOP(retour dans système), SIGN('-'unaire), EXIT et RESET ont un pourcentage inférieur à 0,01%.

Si on regarde la fréquence pure, on trouve 432 occurences de l'instruction BIT dans un programme et son absence dans tous les autres. Les autres instructions sont beaucoup moins nombreuses ($\leq$20).

Les instructions EXIT et RESET sont exceptionnelles, quant à ODD, PRED et SUCC, elles n'ont jamais été rencontrées.

VI.3.b) Résultats particuliers

- 13,1% des affectations sont des affectations à la variable de contrôle utilisée dans les boucles "For".

Cette variable étant une varaible scalaire (sauf réel), l'opération d'affectation est très simplifiée par rapport au cas général.

Un aménagement dans ce cas est très profitable.

- D'autre part, on obtient le résultat suivant pour l'instruction d'aiguillage:

$$\frac{\text{Pourcentage instruction OF}}{\text{Pourcentage instruction CASE}} = \frac{0,439}{0,273} = 1,61$$

Nous voyons donc que le choix de la solution des tests séquentiels fait au chapitre II ne pénalise pas trop l'exécution.

Ceci justifie à postériori le choix de cette option.

Dans les programmes mesurés le "case" comporte trois cas, ceci n'est pas tellement gênant dans la mesure où le nombre moyen de cas dans un "CASE" est statiquement de 3,3.

Remarque:

Si les fréquences de passage dans chacune des 3 alternatives étaient équidistribuées, nous devrions obtenir un rapport de 2 au lieu de 1,81.

Les rapports $\frac{\text{nombre "OF"}}{\text{nombre "CASE"}}$ pour les 5 programmes qui en comportent sont: 2 - 1,87 - 1,46 - 1,17 - 1,6.

VI.3.c) Récapitulatif par catégories d'instructions

Nous allons regrouper ici les pourcentages de plusieurs instructions formant une catégorie homogène.

Nous distinguerons 9 catégories:

- Accès à un descripteur de variable,
- opérateurs
  - opérateurs arithmétiques(+,-,*,/,DIV,MOD,-unaire,Abs)
  - opérateurs logiques ($\neg$,&,|)
  - opérateurs de comparaison(<,$\leq$,>,$\geq$,=,$\neq$,IN)
- littéral (entier, réel, scalaire, booléen , powerset),
- manipulation de descripteur (Index, champ, champw, point),
- affectation,
- choix (If, Do, Cycle, Until, Forup, Fordown, To),
- procédure (Appel, Retour, Alloc, Reset, Procédure d'entrée/sortie)
- branchement (goto, exit).

accès %	opérateur	littéral	desc.	affect.	choix	boucle	proc.	branch.
34,54	15,20	10,21	10,14	9,52	7,93	5,47	4,92	0,33

Les opérateurs eux-mêmes se répartissent comme suit:

opérateurs de comparaison	6,92%
opérateurs arithmétiques	6,69%
opérateurs logiques	1,59%
total	15,20%

La conclusion principale de ces mesures est que le calcul d'adresse (nom interne → adresse mémoire) apparaît dans près de 40% des instructions (34,54% + 4,92%), et sera donc une primitive, c'est-à-dire NOM S,D sera une instruction de la micro-machine.

L'accès complet à une variable ou à un élément de structure représente à lui seul près de la moitié des instructions exécutées (34,54%+10,14%).

En comparaison, les instructions de contrôle occupent une faible place (12,40%). La manipulation de descripteurs de segment pour le contrôle dynamique est finalement peu coûteuse pour l'ensemble du programme, en comparaison de la sécurité qu'elle apporte.

VI.3.d) Influence sur le langage intermédiaire

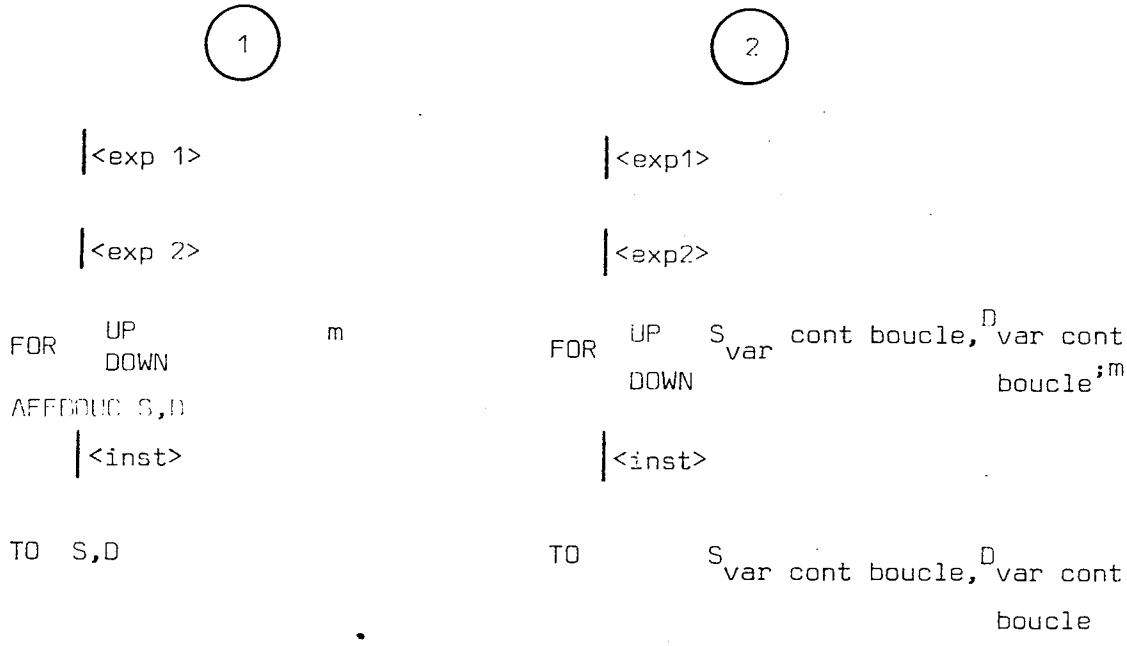
Comme nous l'avons dit plus haut, l'affectation à la variable de contrôle des boucles "For" est très fréquente (1,35%) des instructions du langage intermédiaire.

Deux optimisations du langage sont possibles:

1/ Une instruction spéciale d'affectation à une variable scalaire (sauf réel): AFFBOUC S,D.

Celle-ci pourrait être utilisée aussi dans les affectations normales, mais cela impliquerait un contrôle des types à la traduction.

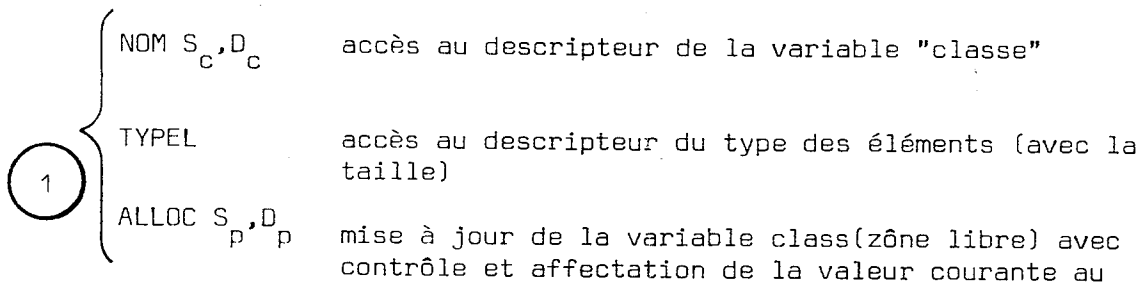
2/ Reporter cette instruction spéciale au niveau suivant en la traitant en "primitive" des instructions FOR et TO, ce qui donnerait une instruction FOR avec un paramètre supplémentaire qui est le nom de cette variable de contrôle. D'où les nouvelles formes:



Des mesures dynamiques plus fines, sur le pourcentage d'affectation scalaire en l'occurance, ainsi que la structure de l'interpréteur permettront de choisir entre ces deux solutions:

Cas de l'instruction Alloc:
La rareté de cette instruction (0,06%) ainsi que sa complexité due au calcul de la taille du nouvel élément, nous amènent à reconsidérer la traduction de cette procédure prédéclarée.

Nous pouvons dissocier les cas où le calcul de la taille est nécessaire. Dans ces conditions nous avons deux traductions:



NOM S_c,D_c

ou (2) APPEL $0,D_t;n$ fonction prédéclarée de calcul de taille

ALLOC S_p,D_p

1 ou 2 remplacent 1 instruction: Alloc $S_p,D_p;n$

VI.4. CONCLUSION

Les choix de primitives pour le langage du niveau suivant est un peu prématuré. Il dépend aussi de l'interpréteur et des mesures dynamiques plus fines seront nécessaires.

Nous avons, cependant, vu plus haut que l'instruction NOM S,D devrait être une primitive.

Il en est de même pour les instructions 'LITT' et 'UNZER' qui représentent plus de 10% des instructions et qui sont simples.

En ce qui concerne les autres instructions, intervient alors un concept de complexité et/ou de relation avec d'autres instructions.

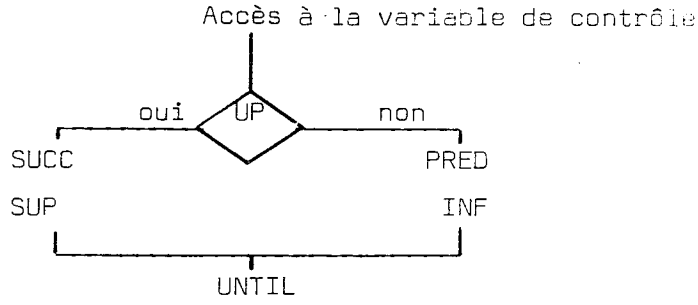
Exemples:

- Dans l'instruction TO figurent sémantiquement les instructions 'SUCC' (ou 'PRED') et 'UNTIL'.

- Le passage de paramètres a de nombreux points communs avec l'affectation d'où vraisemblablement des primitives communes.

A priori, dans le cas du premier exemple, nous avons grossièrement:

TO S,D $\sim$ NOM S,D



Cette instruction sera composée au moins de ces primitives (ces instructions pouvant elles-mêmes être décomposées en primitives).

En bref, le choix des primitives dépend:

- des résultats des mesures dynamiques,
- des relations avec d'autres instructions du langage intermédiaire,
- de la complexité de l'instruction considérée (interprétation).

VII.- INTERPRETEUR

Le choix d'une structure mono ou multi-processeur est laissé à l'appréciation du concepteur.

Il résulte en grande partie d'un compromis coût/performance.

La solution mono-processeur, avec ses implications, ayant déjà été traitée de nombreuses fois par la Compagnie BURROUGHS en particulier [3], nous étudions ici l'emploi de plusieurs processeurs spécialisés.

Le langage et les mesures guident en partie le découpage des fonctions.

Cette option nous permettra des comparaisons sur le langage du niveau suivant.

VII.-1. STRUCTURE GENERALE

VII.1.a) Introduction

A partir de la description formelle (§ V), nous avons regroupé les trois piles en une seule: économie de place et aussi de temps.

Nous disposerons donc maintenant de seulement trois entités distinctes:

- la liste des instructions, disposées dans un tableau (PROG),
- la liste des descripteurs de types, rangés dans un tableau (TYP),
- les descripteurs de variables et de segments ainsi que les valeurs, rangés dans une pile (CTX): la pile de contexte.

Celles-ci sont implantées dans la mémoire centrale.

Classiquement, dans une machine langage, nous disposons d'une pile d'évaluation, implantée en mémoire, avec le sommet et le sous-sommet dans des registres.

Dans le cadre d'une unité de traitement multi-processeur, l'utilisation d'une pile pour l'évaluation, fait perdre tout le bénéfice que l'on peut attendre de la séparation en plusieurs processeurs. D'où l'utilisation d'une file.

D'autre part, le travail sur les descripteurs est très important ; il s'effectue dans deux variables spéciales (DVAR,DP).

A tout cela il faut ajouter les variables de gestion de ces différents éléments. Une contrainte supplémentaire apparaît à cause du regroupement en une seule pile: le besoin d'un séparateur entre les valeurs et les descripteurs de segment. Cette séparation est indispensable pour l'instruction 'EXIT'. De plus, l'utilisation de plusieurs processeurs impose une séparation nette des fonctions (et aussi la duplication de certaines variables).

Nous introduisons une nouvelle instruction "INST", qui est la traduction de la séparation entre les déclarations et les instructions (BEGIN) de la procédure. Cette instruction indique l'entrée dans le segment principal de la procédure. Elle construit un descripteur de contrôle spécial qui contiendra l'adresse de retour. La marque de bloc ne contient plus que les informations nécessaires à la gestion de l'espace des noms.

INST: construit le descripteur 'segment principal', avec l'adresse de retour, et l'empile sur la pile de contrôle (segments).

Avec ce mécanisme, l'instruction 'exit' recherchera le segment de l'étiquette externe seulement jusqu'à ce descripteur.

Cette instruction permet de séparer les concepts de contrôle et d'adressage. Dans le cadre multi-processeur,où il y a des processeurs différents pour chacune de ces notions, ceci facilite l'asynchronisme.

Remarque:

Tous les noms utilisés ici (PROG, TYP,...) sont ceux des différentes entités déclarées dans l'interpréteur écrit en PASCAL donné à l'annexe V.

VII.1.b) Mémorisation

La figure suivante montre l'organisation générale de la mémoire dans le cadre du regroupement en une seule des trois piles formelles décrites précédemment (§ V).

Remarque:

Dans l'hypothèse où il existe un processeur de contrôle autonome, la pile de contrôle devra être séparée pour permettre le maximum d'asynchronisme , d'où un gain de temps (l'instruction INST devient inutile dans ce cas de deux piles séparées).

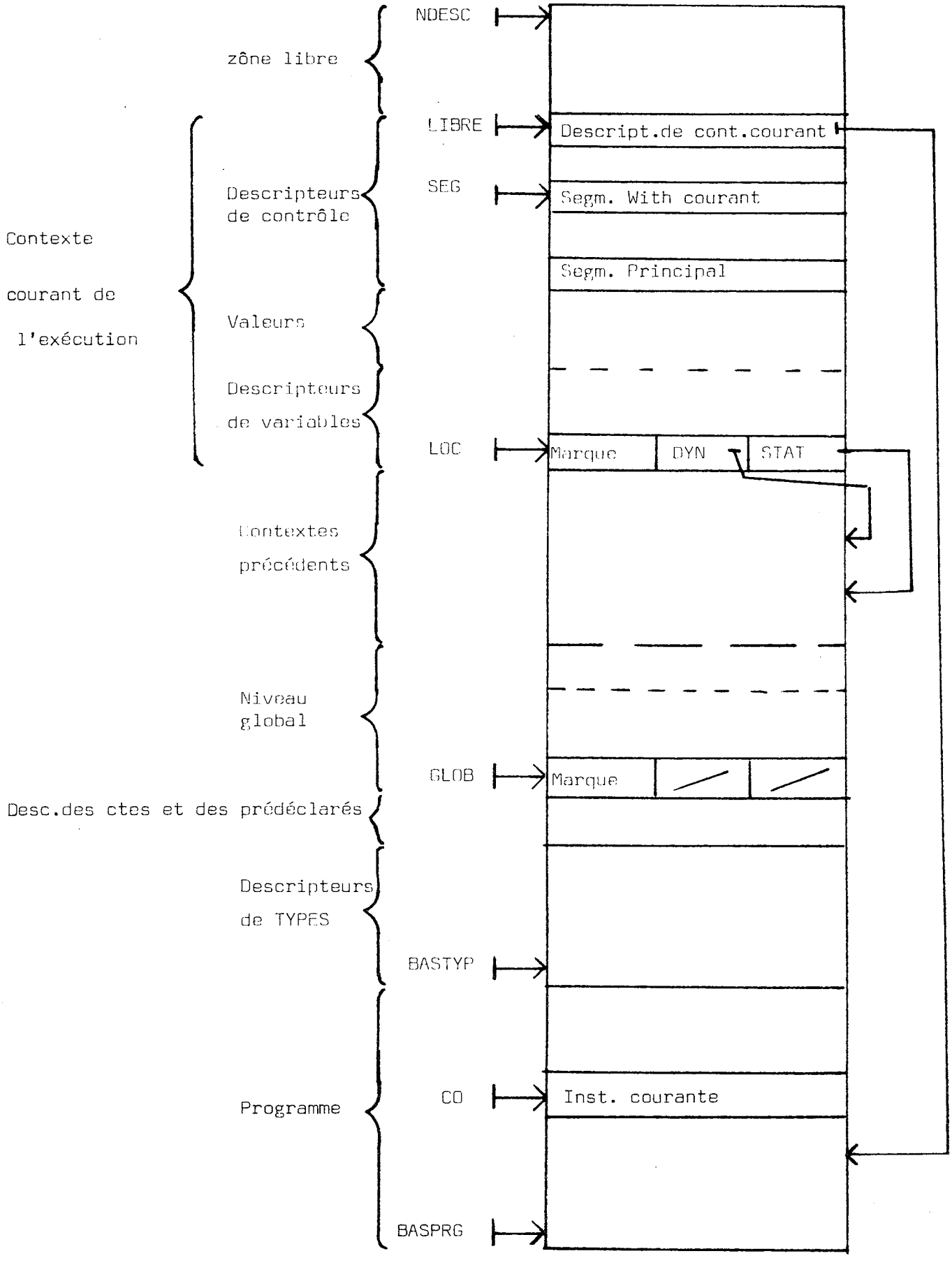


Figure VII.1. ORGANISATION DE LA MEMOIRE

VII.2. ORGANISATION

VII.2.a) Différents processeurs

Un découpage de l'unité de traitement a été effectué en fonction des données manipulées, des opérations à effectuer et des fréquences dynamiques.

Il y a quatre types de données différentes:

- les descripteurs de variables,
- les descripteurs de types,
- les descripteurs de contrôle,
- les valeurs.

auxquels il faut ajouter les instructions.

Au cours d'une opération complète, on peut distinguer 5 phases:

- lecture et décodage des instructions,
- accès aux descripteurs de variables,
- accès aux valeurs (si elles ne figurent pas dans le descripteur),
- opération,
- contrôle et rangement du résultat.

La valeur étant souvent dans le descripteur et l'opération étant en général rapide (pour les plus fréquentes), les deux phases accès valeur-opération peuvent être confondues en une seule.

D'autre part,de nombreuses instructions de contrôle peuvent être exécutées immédiatement (BOUCLE, INST, GO...).

D'où le découpage de l'unité de traitement en quatre processeurs spécialisés:

- Processeur de contrôle:
  - lit et décode les instructions,
  - exécute les instructions de contrôle,
  - gère les descripteurs de contrôle (segments),
  - distribue les autres instructions aux autres processeurs.

- Processeur d'accès:
  - . calcule l'adresse des variables et lit leurs descripteurs,
  - . gère l'espace des noms,
  - . exécute les instructions "littéral",

- Processeur opératif:
  - . exécute les opérateurs (accès à la valeur si elle ne figure pas dans le descripteur et effectue l'opération demandée),

- Processeur rangement:
  - . exécute les instructions d'affectation (contrôle de la valeur et rangement),
  - . exécute les instructions de création (déclaration),
  - . passe les paramètres,
  - . fait l'allocation.

Les charges (tirées des mesures dynamiques) sont très différentes:

- le processeur de rangement exécute 11% des instructions, mais elles sont relativement complexes et lentes.

- le processeur de contrôle exécute 18% des instructions. De plus, il doit distribuer les autres, il doit donc être rapide. Ses opérations sont simples.

- le processeur opératif exécute 25% des instructions. Ses instructions les plus fréquentes sont très rapides (addition d'entiers par exemple).

- le processeur d'accès exécute à lui seul, près de la moitié des instructions (45%), mais elles sont très simples (NOM, LITT...).

Ces quatre processeurs sont en concurrence vis-à-vis de la mémoire.

Remarque:
Il existe des instructions multiplexées sur plusieurs processeurs. Par exemple l'instruction "APPEL" met en jeu les processeurs de contrôle (branchement), d'accès (descripteurs de paramètres) et de rangement (passage des paramètres).

VII.2.b) Synchronisation

En principe ces quatre processeurs sont indépendants.

Nous distinguerons deux types de synchronisations:

- sur le code,
- sur les valeurs.

La première intervient pour synchroniser le processeur de contrôle avec les trois autres processeurs.

Le processeur de contrôle peut se mettre en attente:

- instruction "SORT" appliquée à un segment With:
  le processeur d'accès doit avoir terminé d'utiliser l'adressage particulier à ce segment avant de dépiler le descripteur de contrôle. Elle représente dynamiquement 0,11% des instructions du langage intermédiaire.

- instructions de choix (IF, UNTIL...):
  on doit attendre la fin de création des variables locales à la procédure avant de tenter de les accéder (blocage sur l'instruction "INST"). Elles représentent dynamiquement 1,53% des instructions du langage intermédiaire.

Nous voyons donc que le processeur peut être bloqué au maximum sur 9,98% des instructions (dynamiques).

La synchronisation sur les valeurs se situe entre les trois autres processeurs.

Problème de dépendances:

Ce problème des dépendances est particulier aux trois autres processeurs. Nous y reviendrons brièvement au paragraphe suivant.

Nous arrivons finalement à l'organisation de l'unité de traitement décrite dans le schéma ci-après.

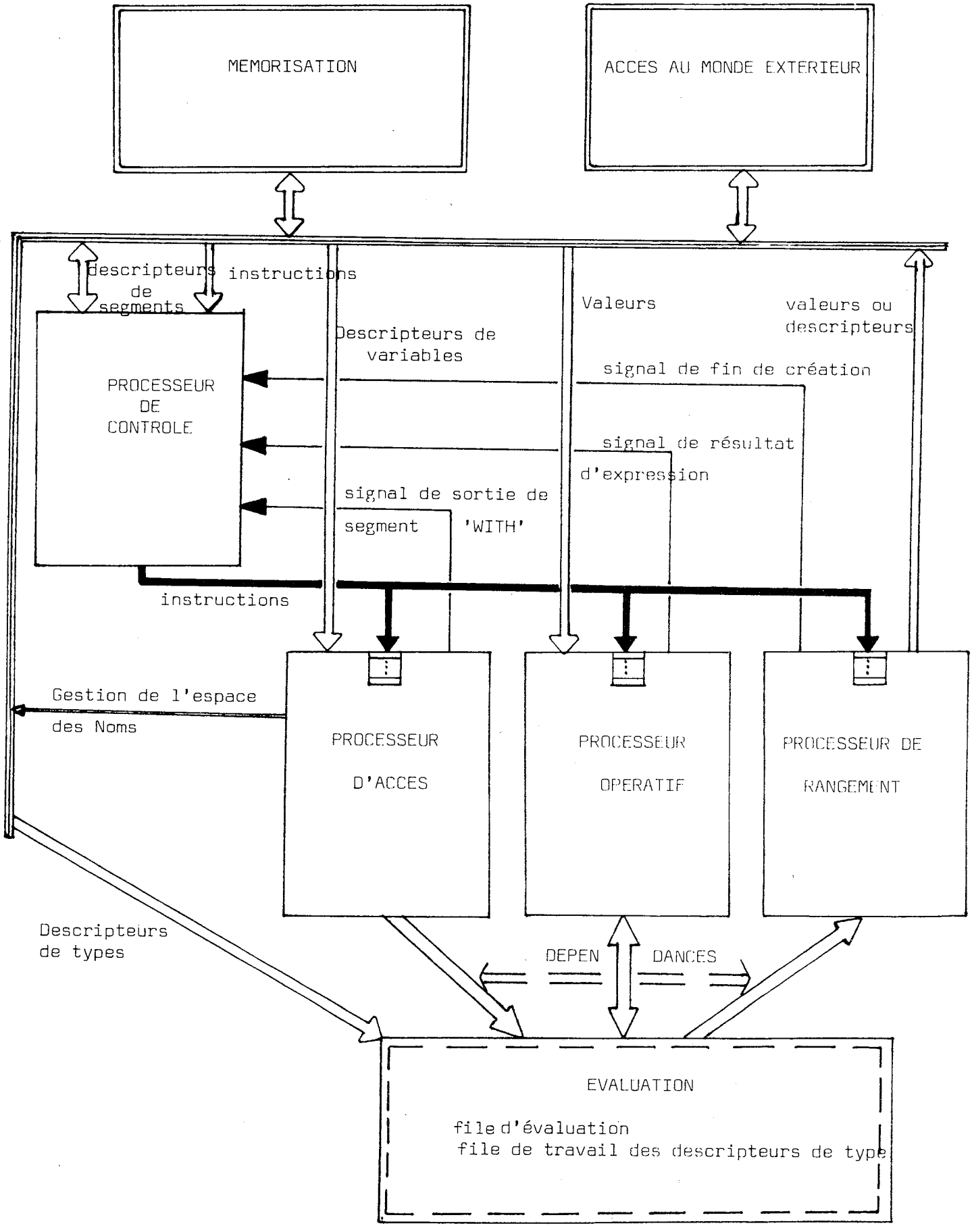


Figure VII.2. ORGANISATION DE L'UNITE DE TRAITEMENT

VII.2.c) Dépendances

L'évaluation des expressions dans un langage évolué est généralement organisée autour d'une PILE contenant les opérandes et les résultats intermédiaires. L'évolution de la pile est définie d'une manière simple par une chaîne d'ordres de forme post-fixée.

Du fait de l'utilisation d'une pile, les trois processeurs précédents ne peuvent pas être désynchronisés: ils sont activés séquentiellement.

Le fonctionnement de la pile est la synchronisation.

Par contre, l'utilisation d'une FILE permet de désynchroniser et d'envisager une organisation "pipe-line" pour l'évaluation des expressions:

1/ le processeur d'accès range les opérandes dans la file,

2/ le processeur opératif extrait les opérandes (préparés par le processeur d'accès) de la file pour évaluer les résultats intermédiaires qu'il place dans la file,

3/ le processeur de rangement effectue l'affectation du résultat final de l'expression (après vérification du type du résultat).

Ces trois processeurs sont en principe, le plus asynchrones possible.

- Le processeur d'accès reçoit du processeur de contrôle les ordres concernant l'accès aux opérandes qu'il range dans la file, à moins qu'il ne se trouve dans la situation suivante: accès à la valeur d'une variable non encore calculée (contrainte de dépendance).

Une dépendance peut être mise en évidence de deux manières:

1°/ Synchronisation au niveau des descripteurs de variables
(solution utilisée dans l'IBM 360/91)

Le descripteur de chaque variable (en mémoire centrale) contient un compteur qui est:

- incrémenté de 1 lorsque la variable apparaît en partie gauche d'une affectation.

. décrémenté de 1 lors du rangement en mémoire de la nouvelle valeur,
. comparé à 0 lors de l'accès à la valeur de la variable (occurence en partie droite).

2°/ Test de la présence d'un descripteur de rangement

Les cas de dépendance sont caractérisés par le fait qu'il existe dans la file un descripteur de rangement placé par le processeur d'accès, mais non encore utilisé par le processeur opératif. Le processeur d'accès doit donc tester la présence dans la file du descripteur de rangement de la variable dont il veut accéder la valeur (occurence en partie droite). Cette solution nécessiterait que la file soit munie d'une fonction de recherche associative des descripteurs de rangement. Or, ces descripteurs peuvent être très dispersés dans la file du fait des "trous" engendrés par le processeur opératif.

La solution proposée consiste donc en l'utilisation d'une autre file dans laquelle seuls les descripteurs de rangement sont stockés ; une recherche dans cette file permet:

1/ de détecter les dépendances sur les variables simples,
2/ de préparer la résolution des dépendances effectuée par le processeur opératif,
3/ de réaliser un "cache" des dernières variables affectées.

- Le processeur opératif opère sur la file dans le même sens que le processeur d'accès: il en extrait ses opérandes (préparés par le processeur d'accès) et y range ses résultats intermédiaires.

Son fonctionnement est moins automatique que dans le cas de l'évaluation sur une pile ; il doit en effet avoir deux pointeurs O_1 et O_2:

- O_2 pointe sur le "sommet" du calcul, il progresse vers le haut d'une position à chaque occurence d'un opérande dans la chaîne de commande ;

- O_1 pointe sur le premier opérande d'une opération diadique situé dans la première place non vide sous le pointeur O_2. Son mouvement peut être:

1/ automatique: recherche séquentielle ou associative de la première place non vide, à l'aide d'un vecteur d'occupation des mots de la file(réalisation hardware coûteuse) ;

2/ calculé: on insère dans la chaîne de commande des ordres spéciaux précisant son mouvement (exemple: recul de r positions).

Globalement, la file est vidée par le bas:

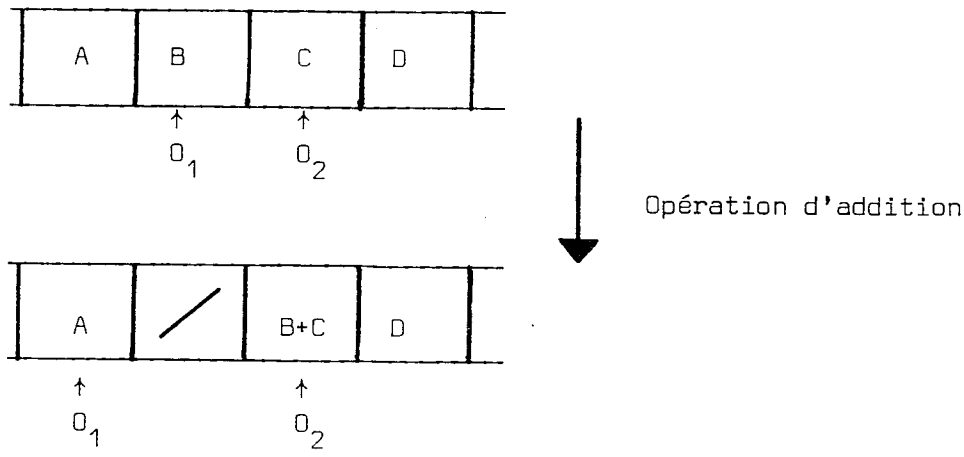
. le résultat d'une opération diadique est mis à la place de l'opérande pointé par O_2 ;
. un "trou" est créé à la place de l'opérande pointé par O_1 qui recule d'une position à la recherche de l'opérande suivant.

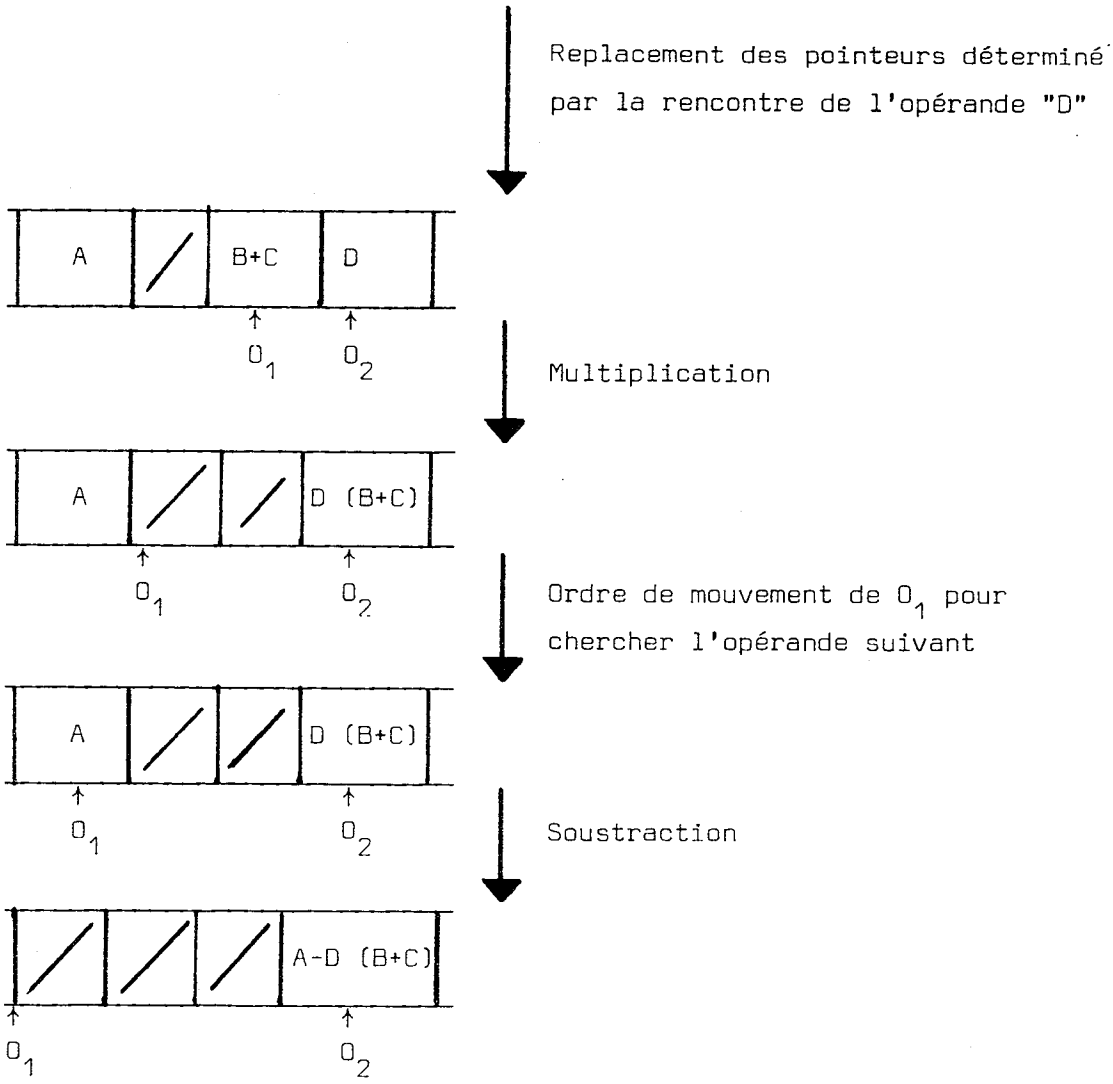
Exemple de fonctionnement

A - (B+C)*D

la chaîne post-fixée est: ABC + D*-

Le processeur d'accès ayant déjà rangé tous les opérandes dans la file, et les pointeurs O_2 et O_1 les ayant suivis, la situation de départ est la suivante:





Les ordres de recherche des opérandes peuvent ici être facilement générés par le processeur de contrôle. Celui-ci glissera parmi les instructions destinées au processeur opératif, les ordres déduits de la forme de l'expression post-fixée qu'il a déjà lue.

- Le processeur de rangement:

La solution (2) du problème des dépendances pour le processeur d'accès nous paraît la plus intéressante.

Dans ces conditions, le processeur de rangement possède une file de variables à affecter.

Le processeur opératif peut transmettre un signal 'fin expression' au processeur de rangement avec la place du résultat de l'expression.

Ce signal est, par ailleurs indispensable dans les instructions du langage intermédiaire.

En effet, lors d'une évaluation sur pile, les résultats de plusieurs expressions consécutives (passage de plusieurs paramètres par exemple) sont contigüs dans la pile.

Mais ici, ces résultats seront distribués un peu n'importe où dans la file d'évaluation.

Une instruction 'FIN EXP' permettant de repérer le résultat courant est indispensable dans quatre cas:

PASCAL: <Nom procédure>(<exp_1>,<exp_2>,..,<exp_n>);
langage intermédiaire: <exp_1>FINEXP<exp_2>FINEXP...<exp_n>APPEL S_p,D_p;n

PASCAL: <Nom tableau>[<exp_1>,...<exp_n>]
langage intermédiaire:<exp_1>FINEXP...<exp_n> INDEX S,D;n

PASCAL:[<exp_1>,...<exp_n>]
langage intermédiaire: <exp_1>FINEXP ...<exp_n> BITn

PASCAL: with <var_1>,....<var_n> do <inst>;
langage intermédiaire: <var_1>FINEXP ....<var_n> WITHn <inst> SORT

L'introduction de cette instruction ('FINEXP') est un exemple de remontée de primitive due à la structure de la machine (interpréteur).

Remarque:

Le processeur opératif est synchronisé sur ce signal pour les instructions d'affectation et le passage des paramètres. Par contre, c'est lui qui synchronise le processeur de contrôle pour les déclarations (§ VII.2.b).

Le schéma suivant décrit le mécanisme d'évaluation.

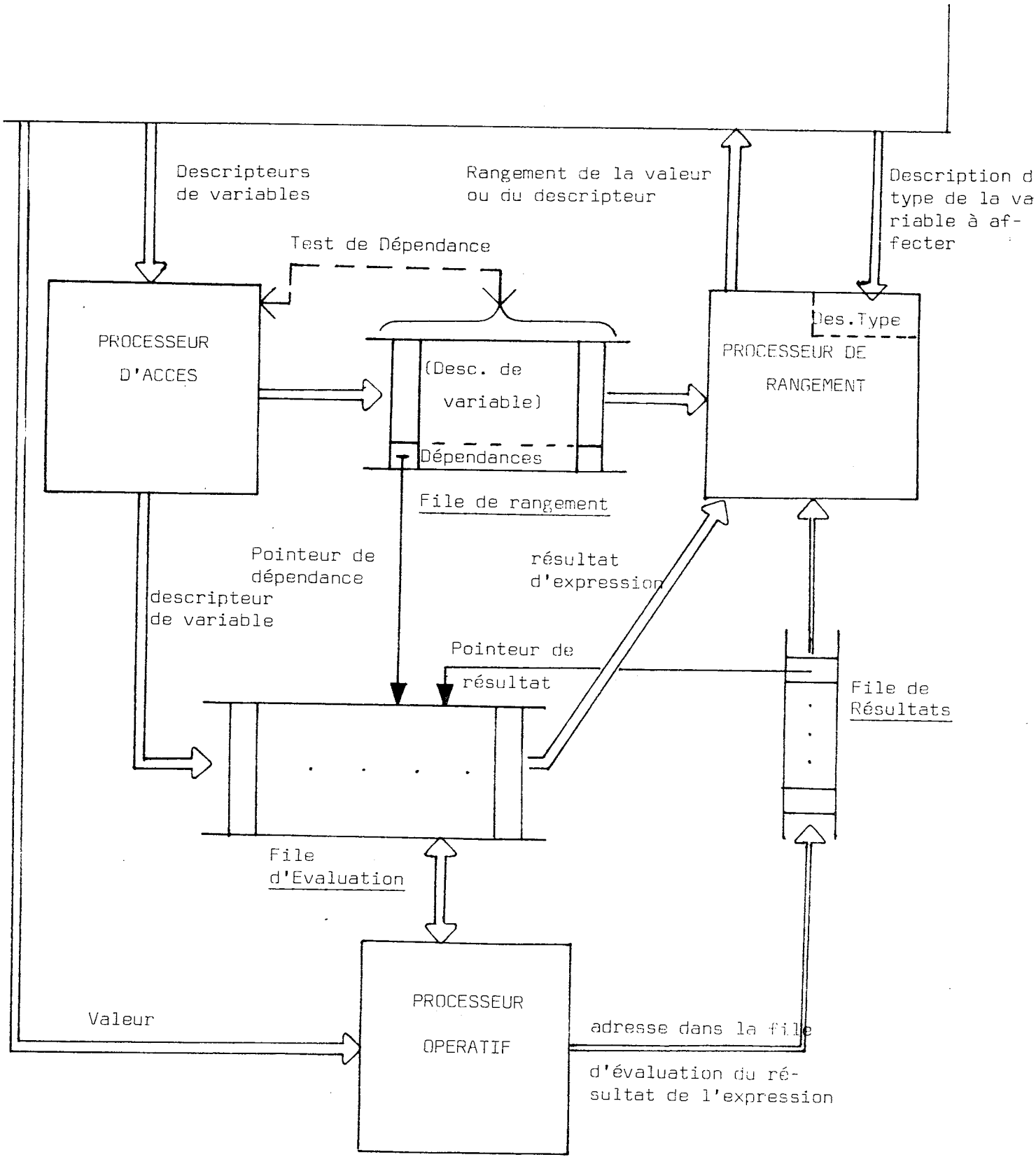


Figure VII.3. ORGANISATION DE L'EVALUATION

VII.2.d) Conclusion

En tout état de cause, comme on pouvait s'y attendre, le choix multiprocesseur introduit beaucoup de primitives fonctionnelles:

. les primitives de synchronisation sur le processeur de contrôle (§ VII.2.b),

. les primitives pour le processeur opératif,

alors que dans le cas mono-processeur et pile, les seules primitives fonctionnelles sont: Empile, dépile et éventuellement échange.

Bien entendu, cette organisation de l'unité centrale influe sur la détermination des primitives du langage intermédiaire.

Un exemple significatif est celui de l'instruction d'appel de procédure:

APPEL S,D: va être transformée en une suite de primitives distribuées aux différents processeurs.

- une primitive NOM S,D pour le processeur d'accès en vue de l'accès au descripteur de procédure, puis, automatiquement, à ceux de paramètres

- des primitives pour la gestion de la pile de contrôle, pour le processeur de contrôle lui-même.

- des primitives de passage de paramètres au processeur de rangement et une primitive lui demandant de signaler au processeur de contrôle la fin du passage de ces paramètres.

(Le calcul des paramètres effectifs a été effectué précédemment par le processeur opératif).

Cette influence du découpage en processus sur la détermination des primitives, impose une simulation en vue de mesures dynamiques du parallélisme (simulation du fonctionnement sur la file d'évaluation ("pipeline").

Actuellement, seules les primitives déjà déterminées au chapitre précédent, plus les instructions purement de contrôle (SORT, FIN, BOUCLE) sans test, en raison de leur simplicité et leur fréquence, peuvent prétendre surement au titre de primitive.

VIII.- EXTENSIONS DU LANGAGE INTERMEDIAIRE

VIII.1. COMPLEMENTS POUR PASCAL

VIII.1.a) "Classe" dans le nouveau PASCAL

Dans un souci de compatibilité , une nouvelle version du langage PASCAL a été développée par JENSEN et WIRTH [WI-74].

La modification essentielle du langage est la disparition du type "CLASS". Les autres modifications par rapport à la version originelle du langage ne sont que des détails.

Une seule de ces modifications a une importance. Elle concerne le mode de passage des paramètres. Les mots clés sont réduits à trois: 'VAR', 'PROC', 'FUNC' qui ont les mêmes significations que dans la version précédente. 'CONST' a disparu ainsi que 'VALUE'. L'absence de mot clé indique le passage par valeur pur. C'est-à-dire que dans tous les cas (même pour une structure) il y a recopie.

En ce qui concerne les types 'class' et pointeur, seul subsiste ce dernier.

La nouvelle déclaration se présente sous la forme:

<Decl typ pointeur> ::= <nom type pointeur> '=' '↑' <type>

<type> : déclaration de type autre que fichier.

Une variable d'un tel type sera un pointeur sur un élément de type <type>.

Ceci se traduit dans l'implémentation proposée par l'utilisation d'une seule "classe" qui contiendra tous les éléments de tous les types déclarés et dont la taille est indéfinie.

La procédure de gestion de cet espace est nommée 'NEW'. Elle provoque l'empilage dynamique d'un nouvel élément indépendamment de son type.

La conséquence immédiate d'un tel mécanisme est le mélange dans le "tas" de structures différentes. La possibilité de fonctionnement en pile de plusieurs variables "classe" est perdue.

C'est pourquoi dans cette version du langage il n'existe plus de procédure "Reset".

Tout se passe comme si une "classe" unique pouvant contenir des éléments de tous les types possibles (sauf fichier) avait été pré-déclarée, dont la taille soit infinie et sur laquelle l'opération de dépilage soit interdite.

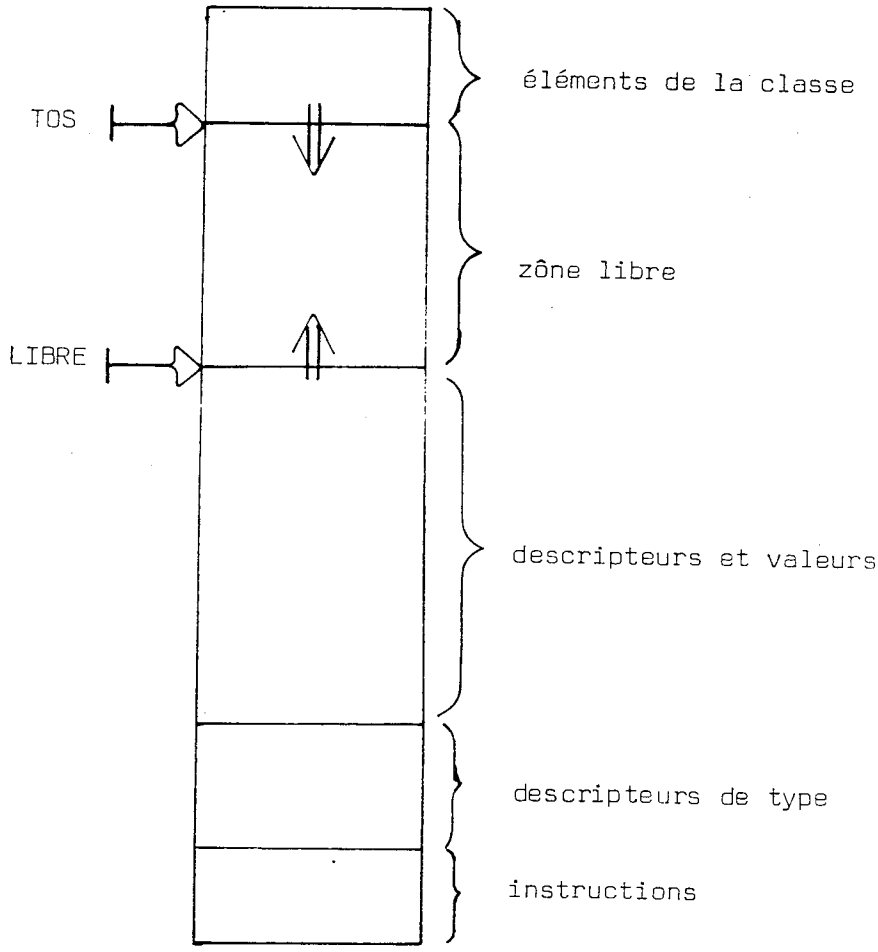
C'est en quelque sorte le tas ("Heap") utilisé dans l'implémentation d'ALGOL 68 pour les générateurs globaux.

La procédure NEW est, exception faite de ceci, absolument semblable à Alloc.

Ceci simplifie en fait l'interprétation car il n'y a plus besoin de descripteur de classe, ni de spécifier le nom de la classe pour l'instruction Alloc.

Le contrôle de dépassement de capacité de la classe est devenu inutile. Seul reste le contrôle de dépassement de capacité de la zône mémoire allouée pour le programme.

L'implémentation de cette classe unique est:



Indépendamment du problème syntaxique de compatibilité avec l'ancienne version du langage, nous pouvons intégrer ce nouveau concept de classe à l'ancien.

Il suffit pour cela de prédéclarer une variable 'classe' spéciale, dont le descripteur précise la particularité.

"classe"	"0"	adresse "libre" courante dans le tas

cette taille nulle
indique la particularité de la
variable

Le type d'un élément de cette classe est spécifié dans la variable de type pointeur, comme précédemment.

La procédure d'allocation est ainsi conservée presque intégralement. Dans le cas présent, seule change la partie de l'algorithme qui vient après le calcul de la taille de l'élément.

```
if DVAR,PT = 0 then          (DVAR contient la description de classe)
   /* classe unique */
   begin
       if DVAR.PV - Taillecalculée < Libre then
          begin  /* dépassement de capacité */
          DP.PV := NIL ; (DP contient descripteur de pointeur)
        FICHER  := 'TAS PLEIN' ; Put (FICHER)
            end else begin
        DP.PV := DVAR.PV ;   /*
          DVAR.PV := DVAR.PV - Taillecalculée /*mise à jour de libre*/
                     end
    else begin /*cas normal*/                end ;
```

Cette modification complique légèrement l'instruction d'allocation ; mais elle permet, à peu de frais, de rendre le nouveau PASCAL compatible avec la version actuelle du langage intermédiaire.

VIII.1.b) Fichiers séquentiels

La notion de fichier dans PASCAL, notamment avec les procédures "get", "put", "eof" est dépendante de la machine CDC sur laquelle les premières implémentations ont été faites.

C'est pourquoi nous nous appuierons ici sur la définition axiomatique, proposée plus tard par WIRTH [WI-72], des fichiers séquentiels.

A partir de cette définition axiomatique, nous déduisons que:

- un fichier séquentiel est une file particulière d'éléments tous de même type dont un seul est accessible à un instant donné par l'intermédiaire d'un pointeur.

- cette file est parcourue dans un sens unique, élément par élément ;

- les opérations permises sont: l'avancement à l'élément suivant avec lecture ou écriture (fichier "IN" ou "OUT" ; procédure "get" ou "put" et retour au début ;

- la déclaration de fichier précise outre le type des éléments, le mode d'utilisation (entrée ou sortie).

Nous avons une remarque à formuler: le mode d'utilisation précisé implicitement en utilisant get ou put est redondant avec la déclaration.

La définition axiomatique du fichier comme une pseudo-file, et sans tenir compte de la contrainte due au périphérique, nous induit à réduire les procédures get et put à une seule procédure adaptée à la notion de file et où il n'y a plus redondance du mode: 'SUIVANT(f)'.

La procédure Reset conserve son sens de retour au début du fichier, avec de plus une modification éventuelle de ce que nous appelons le mode du fichier (entrée-sortie).

Pour différencier cette procédure de celle appliquée à la classe, nous l'appellerons DEBUT (f, $\begin{cases}\text{'IN'}\\\text{'OUT'}\end{cases}$) ;

Une possibilité supplémentaire peut être apportée au langage par l'introduction d'une procédure IGNORE (f) qui passe à l'élément suivant sans aucune autre opération (elle est indépendante du mode du fichier).

VIII.1.c. Fichier direct

L'équipe IREP-CICG a introduit dans son implémentation de PASCAL la notion de fichiers directs comme des pseudo-tableaux (fichier régional 1 d'OS/MVT).

Les procédures d'accès ont la forme:

```
get (f, <exp> )
put (f, <exp> )
```

L'expression est considérée comme un index dans la variable fichier direct.

En accord avec la déclaration de tableau de PASCAL, nous obtenons alors la déclaration suivante:

<var fichier>[<mode>]: file direct[<type index>]of <type>

D'où une information: 'Direct'/'Séquentiel' dans le descripteur de type fichier.

Une contrainte mineure peut être imposée: la borne inférieure du type de l'index doit être nulle.

Dans ces conditions le descripteur de type fichier contient seulement la borne supérieure du "tableau".

D'où le format de descripteur décrit dans l'annexe IV.

L'opération d'accès est alors:

INDEX (f, <exp>)

l'élémentde fichier est toujours désigné par f↑.

Nous n'avons pas ici d'opération équivalente à DEBUT puisque le retour au début est sans signification.

D'où une opération MODE (f, <mode>) qui va modifier le descripteur de la variable fichier.

Remarque:

Nous pouvons sortir maintenant la modification éventuelle de mode dans la procédure 'début' pour les fichiers séquentiels. La procédure MODE est indépendante du type du fichier.

VIII.1.c) Déclaration de fichier

Le traitement des fichiers séquentiel et direct décrits précédemment doit être étendu à leur déclaration, en accord avec les déclarations des variables d'autres types.

La déclaration en mode "IN" supposant la pré-existence du fichier, le problème se pose surtout pour les fichiers déclarés en mode "OUT" qui n'existent pas encore à ce moment.

- fichier séquentiel

C'est une pseudo-file dont le fonctionnement est comparable à une classe. La déclaration d'une classe initialise le descripteur à la valeur "classe vide". Nous ferons donc de même pour les fichiers séquentiels.

L'opération de création d'une variable fichier séquentiel doit , outre créer le descripteur correspondant, mais aussi créer un fichier vide s'il est déclaré en mode 'OUT'(par un appel à une procédure système).

- fichier direct

Il est assimilé à un tableau et lors de la création du tableau ses éléments sont mis à la valeur "non initialisé".

L'extension au fichier direct est immédiate. Lors de la création du descripteur de type fichier, les éléments dont le nombre est spécifié dans la déclaration seront alloués avec la valeur "non initialisé".

Ceci peut être réalisé pratiquement en écrivant le nombre spécifié d'enregistrements "muets" dans le fichier direct.

Remarque:

Ceci est fait lors de l'ouverture d'un fichier Régional 1 déclaré UPDATE et qui n'existe pas encore dans OS/MVT.

L'instruction de déclaration d'un fichier de mode 'IN' vérifie l'existence du fichier.

VIII.1.d) Conclusion sur les fichiers

En résumé les opérations possibles sont les suivantes:

Nom	type	mode	paramètres	effet
SUIVANT	Seq	IN/OUT	nom fichier	accès à l'élément suivant(IN→lecture, OUT→écriture)
DEBUT	Seq	IN/OUT	nom fichier	repositionnement au début du fichier
IGNORE	Seq	IN	nom fichier exp entière	saut de <exp entière> éléments
MODE	Seq/Dir	IN/OUT	nom fichier,mode	impose le mode précisé
INDEX	Dir	IN/OUT	nom fichier exp entière	contrôle de borne et accès au <exp entière>^{e} élément (IN→lecture, OUT→écriture)
EOF	Seq	IN	nom fichier	retourne valeur boolèenne 'fin de fichier'

Au niveau du langage intermédiaire l'instruction de création est conservée C'est "CREE n,DV". DV étant ici la concaténation du préfixe

"fichier ou $\begin{matrix}\text{entrée}\\\text{sortie}\end{matrix}$ " et du nom interne du descripteur de type.

Remarque:

Dans la terminologie IBM, le mode "UPDATE" pour les fichiers, indique que l'écriture et la lecture sont possibles, le fichier existant déjà.

La simulation PASCAL de ce mode est une déclaration en mode entrée ('IN') et l'utilisation chaque fois que c'est nécessaire de la procédure MODE avant la procédure d'accès.

Bien entendu des procédures plus puissantes peuvent être définies à partir de ces opérations, par exemple:

LIRESEQ(f) ≡ MODE(f,IN) ; SUIVANT(f)

Remarque:

Il existe un cas particulier de fichiers. Ce sont les fichiers standards d'entrée (carte) et sortie (imprimante).

En PASCAL ce problème est résolu en les prédéclarant:

```
SYSIN[IN]: file of array[0...79]of char ;
SYSPRINT[OUT]: file of array[0..131]of char ;
```

On peut, de plus, prédéclarer une procédure d'accès pour chacun de ces fichiers qui utilisera des fonctions de conversions et éventuellement de formats.

Dans un premier temps, la traduction des opérations définies plus haut pour les fichiers sera conservée, c'est-à-dire un appel à une procédure prédéclarée.

La définition d'un système, pour les entrées-sorties de la future machine, permettra de définir une solution plus optimale.

Vraisemblablement, l'appel d'une procédure pré-déclarée pour chacune des six opérations, sera remplacé par une suite d'une ou plusieurs instructions, leur fréquence dynamique étant non négligeable (0,36% de get ou put). MODE (f,<mode>) peut facilement être traduit en une seule instruction, étant donné sa simplicité.

VIII.2. Langages de type ALGOL

Nous ne nous attachons pas à définir ici une machine ALGOL 68 ou PL/1, mais à étudier quelques concepts importants de ces langages qui ne figurent pas dans PASCAL.

Bien entendu, l'obtention d'un langage intermédiaire pour ALGOL 68 ou PL/1 serait différente dans ses détails, mais ces langages ayant de nombreux points communs, la forme générale serait conservée.

Nous allons donc indiquer quelques modifications indispensables pour ces langages, ces modifications étant minimum. D'autres propriétés sémantiques de ces langages (collatéraux d'ALGOL 68 ou sémaphore, variables externes de PL/1...) n'étant pas étudiées ici parce que trop éloignées de PASCAL.

L'ensemble des instructions des langages traités ici est donc un aperçu du traitement à faire subir à PL/1 ou ALGOL 68.

VIII.2.a) Déclarations-structures de données

PASCAL a une structure procédurale contrairement à ALGOL 68 et PL/1. Aucune déclaration n'est possible ailleurs qu'au début de la procédure, alors que dans les autres, des déclarations sont possibles dans les blocs internes à la procédure.

La solution la plus simple, qui conserve la validité des mesures par niveau lexicographique, est de laisser au compilateur le soin de ramener ces déclarations au début de la procédure qui contient le bloc (technique habituelle utilisée dans les compilateurs).

Des mesures statiques de ces déclarations sont nécessaires pour justifier ce choix.

Les structures de données de base de ces trois langages offrent de nombreux points communs: entiers, réels, booléens, tableaux statiques, pointeurs (références en ALGOL 68), structures.

La différence entre un pointeur et une référence d'ALGOL 68 tient au fait que la référence ne se fait pas uniquement dans le "tas", mais peut aussi être l'adresse d'une variable dans la pile de contexte.

Le "record" de PASCAL sans aiguillage représente les structures. Les variables "Based" avec leur pointeur de PL/1 peuvent être simulées dans la classe prédéclarée, avec les pointeurs PASCAL. Chaque procédure "Allocate" étant une procédure "alloc" de PASCAL ne comportant pas de classe.

La simulation des variables "controlled" est plus compliquée en raison du fonctionnement en pile.

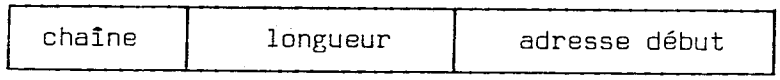
```
DCL A fixed bin (15,0) CTL ;↔ T=  record  A : short integer
                                          C : ↑T end /*chaînage*/
                                P : ↑ T ; P := NIL ;
chaque référence à A est traduite par P↑ .A
Allocate A devient Q:=P Allocate(P) ; P↑ .C:=Q;
Free P      devient P:=P↑ .C
```

Le compilateur a donc ici un gros travail à faire. Il en est de même pour les générateurs globaux d'ALGOL 68.

Par contre, les chaînes de caractères ou de bits sont très différentes finalement d'un tableau à une dimension de caractère ou de powerset respectivement, surtout dans le cas où elles sont déclarées "VARYING" (PL/1) ou "Flexible" (ALGOL 68).

Ces structures nécessitent donc des descripteurs particuliers et donc des instructions d'accès et affectations originales.

D'après les déclarations les descripteurs auront la forme:



Le descripteur (mode) contient la longueur maximum pour les variables "Varying" ainsi que le type des éléments.

Le concept de tableau à bornes calculées est absent de PASCAL, contrairement aux autres langages de type ALGOL.

La solution est de créer un descripteur à l'interprétation. D'où nécessité de prévoir une nouvelle instruction du langage intermédiaire pour créer le descripteur de tableau.

Le nombre de dimensions du tableau est connu à la traduction, la place dans la table des types, nécessaires pour contenir ce descripteur, peut être réservée à la traduction, et le nom interne de ce type est parfaitement déterminé.

Cette nomination et réservation de place permettent ensuite de traiter ce tableau comme un tableau standard de PASCAL.

L'instruction de création de descripteur de tableau a donc la forme:

CREERTAB <nombre de dimensions>, <nom interne>;<nom type elt>

et la traduction de la déclaration:

```
T = array [-1..N, 0..M] of <type>
    Litt -1
    NOM S_N, D_N
    UNZER 0
    NOM S_M, D_M
    CREERTAB 2, <nom interne>;<nom interne type élément>
```

L'algorithme de cette instruction comporte le calcul du pseudo vecteur à partir de la taille des éléments (par descripteur de type de ces éléments accédés grâce au troisième paramètre de l'instruction) et des bornes évaluées précédemment par la formule du chapitre II, puis la mise de la taille totale et le nom des éléments dans le descripteur de tableau.

Remarque:

La distinction entre tableau statique et tableau dynamique existe dans le langage MARY.

De plus, PL/1 possède le concept de type étiquette. Le descripteur d'étiquette existe dans PASCAL pour les étiquettes externes, mais ce sont des constantes.

La différence se situe donc au niveau du littéral et de l'affectation.

VIII.2.b) Instructions

Le concept de valeurs multiples, pour les procédures (fonctions) ou de sous tableau, n'existe pas non plus en PASCAL.

De nouvelles instructions du langage intermédiaire sont nécessaires. Dans le second cas, le pseudo vecteur qui précise le pas entre les éléments facilite la traduction.

L'absence d'index peut être traduite par une instruction de construction d'un pseudo descripteur prenant en compte le nouveau pas entre les éléments de ce sous tableau (qui sont eux-mêmes des tableaux, le pas devenant alors la taille d'un élément).

Utilisation des étiquettes-littéral

En effet, le problème ne se situe pas au niveau de l'affectation pour les

variables étiquette mais à celui du littéral.

Il n'existe pas de littéraux étiquette en PASCAL et un tel littéral est indispensable si on utilise le descripteur d'étiquette externe du langage intermédiaire.

La forme de ce littéral, pour être compatible avec les formats définis pour PASCAL, est:

Litt 'étiq' , <adresse dans segment>,<nom segment>

L'instruction de branchement est toujours le EXIT S,D défini pour PASCAL puisque le descripteur est conservé.

Au cas où la variable étiquette serait un élément de tableau, une deuxième forme de l'instruction est nécessaire "EXIT" sans paramètre de nom interne.

goto A [I] est traduit en: NOM S_I,D_I

INDEX S_A,D_A

EXIT

L'instruction EXIT n'a pas de paramètre puisque l'élément de tableau (étiquette) est déjà accédée par l'instruction INDEX.

VIII.3.c) Instructions de contrôle

L'instruction aiguillage ("case") d'ALGOL 68 est un cas particulier de celle de PASCAL.

case <exp> of <$inst_1$>;...<$inst_n$> esac ; ALGOL 68

case <exp> of 1:<$inst_1$>;... n:<$inst_n$>end ; PASCAL

Dans ce cas, le tableau de correspondance étiquette→adresse du chapitre II est plus intéressant.

Mais il existe une seconde forme du "case" d'ALGOL 68.

case <exp> of <inst$_1$>;....<inst n-1>; out<inst$_n$> esac

dont la traduction en langage intermédiaire de PASCAL est:

```
|<exp>
CASE m
OF 1, m1
  |<inst 1>
FIN
  :
OF n-1, mn-1
  |<inst n-1>
FIN
  |<inst n>
SORT
```

ou en utilisant la forme déclarative

```
|<exp>
CASE m
Litt m1
Litt mn-1
Litt On-1
OF
 |<inst 1>
FIN
 :
 |<inst n>
SORT
```

Sinon un test de dépassement de "bornes" de l'expression est nécessaire pour le tableau de correspondance, qui s'il est vrai, provoque le branchement à la dernière alternative.

Les boucles "For" sont encore plus différentes, puisque, en ALGOL 68, la variable de contrôle de boucle n'est pas déclarée.

De plus en ALGOL 68 et PL/1, un pas peut être spécifié.

Le descripteur de segment est donc beaucoup plus complexe:

for<var> from <exp$_1$> by <exp$_2$> to <exp$_3$> do <inst> ;

est traduit en:

```
|<exp1>
|<exp2>
|<exp3>
FOR  m  1er test, si vrai construit un descripteur de segment compor-
                  en plus la variable de contrôle de boucle et limite
↱ AFFBOUC         affectation à la variable de contrôle de boucle
|
|  |<inst>
|
└- TO             incrément, test et boucle éventuelle
```

La clause DO<inst> d'ALGOL 68 ou le groupe do de PL/1 sont représentés par:

```
BOUCLE
|<inst>
CYCLE
```

Les instructions "while" et "if" sont presque identiques pour ces trois langages, à l'exclusion de leur portée.

VIII.2.d) Conclusion

Ce bref aperçu, met en lumière la spécificité des langages intermédiaires. La définition d'un langage intermédiaire pour PL/1 ou ALGOL 68 peut quand même utiliser de nombreux résultats de PASCAL, notamment pour les mesures.

Les quelques exemples ci-dessus permettent d'accélérer la démarche pour ces langages, une partie importante de chacun d'eux étant déjà traitée, dans la démarche PASCAL, et les paragraphes précédents.

CONCLUSION

La démarche descendante effectuée sur le langage PASCAL n'étant pas terminée, il serait prématuré de conclure sur la machine PASCAL.

Les concepts dégagés pour la première phase d'une telle démarche sont par contre définitifs et rien n'interdit, à priori, d'appliquer les méthodes procédurales et opératives à d'autres langages non de type ALGOL.

Mais nous voyons qu'une part importante des choix à effectuer est laissée à l'appréciation du concepteur et résulte d'un compromis rapidité-coût. Les contraintes technologiques n'apparaissent pas encore à ce niveau.

Les mesures occupent une place majeure dans la définition d'une telle machine. Les mesures statiques déterminent les formats et codages des données et des instructions, les mesures dynamiques déterminent les primitives du langage intermédiaire.

La hiérarchie de mémorisation, ainsi que la détermination des primitives des différents niveaux seront, elles aussi, en grande partie déterminées par des mesures dynamiques plus fines. Le format de ces primitives (micro-instruction) pourra être influencé par des mesures statiques quoique dans une moindre mesure, des contraintes technologiques apparaissant pour les niveaux suivants.

Le choix d'éclater l'unité de traitement en plusieurs processeurs spécialisés introduit la nécessité supplémentaire de simuler le parallélisme, pour mettre au point les synchronisations (primitives fonctionnelles) et confirmer (ou infirmer) le découpage en ces quatre processeurs. Ces primitives fonctionnelles seront plus nombreuses que dans le cas monoprocesseur, et le microcodage, indépendamment des synchronisations, sera très différent de celui obtenu pour les machines langages classiques.

Un autre aspect de la machine est les relations avec le monde extérieur pour les fichiers et son exploitation.

Un "système" pour la future machine PASCAL reste à définir. Ceci pourrait être résolu dans le cadre des"systèmes hiérarchisés" [AN-73]. Il nous paraît important, pour les fichiers, d'inclure les opérations définies au chapitre VIII.1.

Dans cette optique, les "procédures prédéclarées" d'accès au monde extérieur seraient remplacées par des instructions du langage intermédiaire. Mais nous rentrons alors dans le cadre des "machines système".

Un autre axe de recherche est de définir à partir de la démarche effectuée sur PASCAL et les particularités sémantiques des autres langages de type ALGOL, une machine multilangage.

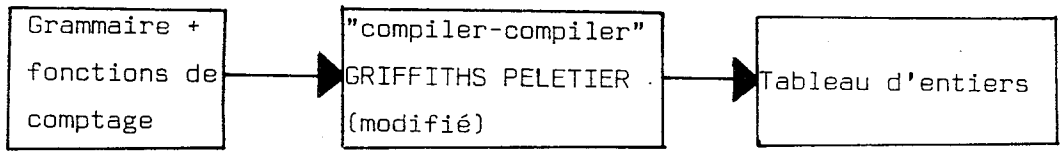
Pour cela une étude plus précise d'ALGOL 68 et PL/1 est nécessaire. Le résultat étant bâti sur le même principe que celui utilisé dans la B1700 . Les nombreuses similitudes entre ces langages permettent de plus d'utiliser un même noyau d'interpréteur pour ces trois langages et un interpréteur complément particulier aux instructions originales de chacun d'eux.

ANNEXES

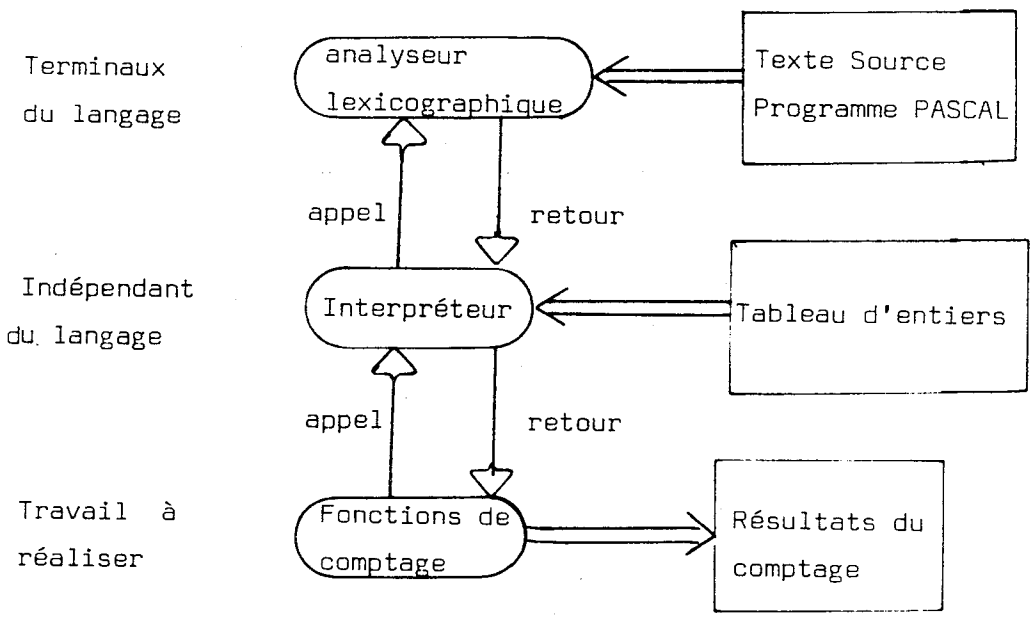
ANNEXE I

SUIVEUR SYNTAXIQUE

Cet outil a été construit à partir de la grammaire du langage dans laquelle ont été insérés des appels à des fonctions de comptage. Cette grammaire a été transformée en un tableau d'entiers, qui conserve toute l'information initiale sous une forme plus compacte, après vérification et éventuellement transformation pour qu'elle soit LL(1).



Structure générale de l'interpréteur:

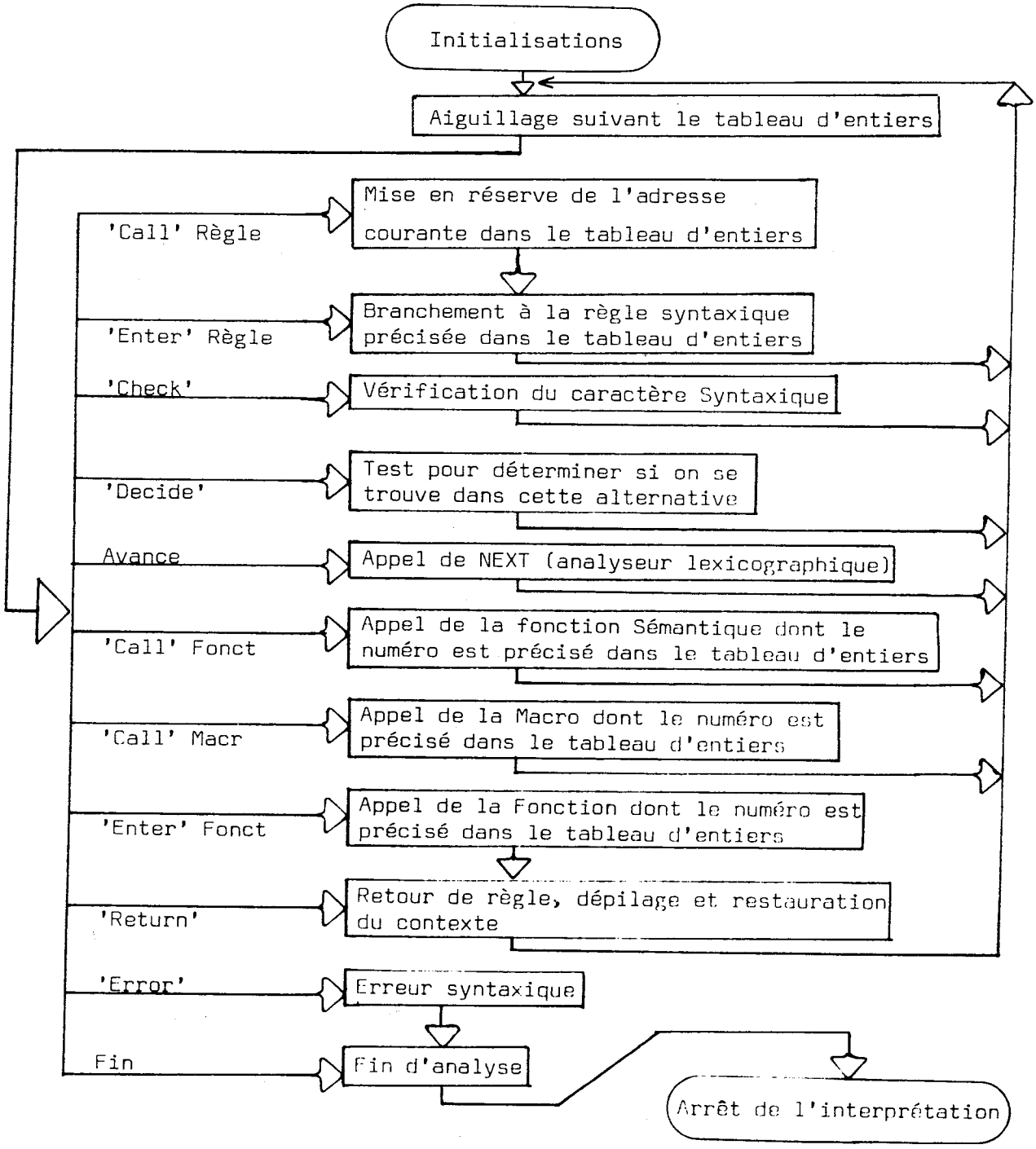


Le module interpréteur est complètement indépendant du langage et du travail à réaliser, il se contente de vérifier la syntaxe spécifiée dans le tableau d'entiers. Le module analyseur lexicographique est dépendant des terminaux du langage, mais indépendant du travail à effectuer. Il fournit l'unité syntaxique suivante (codée). Le module fonction contient les algorithmes des fonctions insérées dans la grammaire.

Cette structure nous permet donc de réaliser le traducteur en ne réécrivant que le module Fonction et en insérant les nouveaux appels correspondants, dans la grammaire.

En fait, le module fonctions peut lui-même être composé de plusieurs modules équivalents (deux modules standards prévus: 'FONCTION' et 'MACRO').

Schéma fonctionnel de l'Interpréteur



A N N E X E I I

SYNTAXE ET FONCTIONS SEMANTIQUES DU TRADUCTEUR

```
<PROG>          *INITAB* &INIMACRO <BLOCK> &GENSTOP& 'PT'

<BLOCK>         <LABELDEC> <CONSTDEC> <TYPDEC> <VARDEC> <VALUDEC> <PROFTDEC>
                                        *FINDEC* 'BEGIN' <TAIL> 'END' *FINBLOC*

<LABELDEC>      'LABEL' <EXITLABEL> 'PVG'
                ()

<EXITLABEL>     &EXTERN& 'INTEGER' <S1>

<S1>            'VG' <EXITLABEL>
                ()

<CONSTDEC>      'CONST' <CONSTLIST> 'PVG'
                ()

<CONSTLIST>     <IDENT> *NOMCONST* 'EG' <SIGNCONST> <S2>

<S2>            'VG' <CONSTLIST>
                ()

<TYPDEC>        'TYPE' <TYPLIST>
                ()

<TYPLIST>       <IDENT> *NOMTYP* 'EG' <TYPDEF> 'PVG' <S3>

<S3>            <TYPLIST>
                ()

<VARDEC>        'VAR' <VARLIST>
                ()

<VARLIST>       <IDELLIST> 'DPT' <TYP> *TYPVARS 'PVG' <S4>

<S4>            <VARLIST>
                ()

<IDELLIST>      <IDENT> *VCLAS* <S40>

<S40>           'VG' <IDELLIST>
                ()

<IDLIST>        <IDENT> *SCALIV <S5>

<S5>            'VG' <IDLIST>
                ()

<PROFTDEC>      <PROCDEC> <PROFTDEC>
                <FONCDEC> <PROFTDEC>
                ()

<PROCDEC>       'PROCEDURE' <IDENT> *CTPROC* <PROCTAIL>

<PROCTAIL>      'PAR1' <PARAMLIST> 'PAR2' 'PVG' <S21> 'PVG'
                'PVG' <S21> 'PVG'
```

```
<FUNCDEC>       'FUNCTION' <IDENT> *CTFUNC* <S20> 'PVG'

<S20>           *PRECDEC* 'PVG' <S21>
                'PAR1' <PARAMLIST> 'PAR2' 'OPT' <IDENT> *TFUNC* 'PVG' <S21>

<S21>           *FORW* 'FORWARD'
                *DEFPROC* <BLOCK> &GENRET&

<TAIL>          <ETIQL> <STATEMENT> <STATLIST>

<STATLIST>      'PVG' <TAIL>
                ()

<ETIQL>         &DEFETIQ& 'INTEGER' 'OPT'
                ()

<VALUEDEC>      'VALUE' <INIT>
                ()

<INIT>          <IDENT> *ACID* 'EG' <VALEUR> 'PVG' <S7>

<S7>            <INIT>
                ()

<VALEUR>        <SIGNCONST>
                *AFFMUL* 'PAR1' <LITTERALIST> *GENAFM* 'PAR2'

<LITTERALIST>   <LITTERAL> <S8>

<S8>            'VG' &INCRLIT& <LITTERALIST>
                ()

<LITTERAL>      <SIGNCONST> <DUP>

<DUP>           *FACMUL* 'MUL' <SIGNCONST>
                ()

<CONSTEL>       &LITNUMB& <NUMBER>
                'QUOT' &INITCHARS& <CHARLIST> &LITCHAING& 'QUOT'

<SIGNCONST>     <SIGN> *FACNUM* <NUMBER>
                'QUOT' <CHARLIST> *FACHAING* 'QUOT'
                <IDENT> *IDCONST*

<INCR>          <SIGN> &INCS& <NUMBER>
                'QUOT' &INCSS& 'CHAR' 'QUOT'

<CONSTANT>      <INCONST>
                &INCSS& <IDENT>

<TYPEL>         'RECORD' <IDENT> *TRCINT*
                *TSCAL* 'PAR1' <IDLIST> 'PAR2' *FINSCAL*
                *TSET* 'POWERSET' <SOMSTYP>
                'ARRAY' *TTABLE* 'CROC' <INDTYPLIST> 'CROSS' 'OF' <COMPTYP> *FINTAB*
                'RECORD' *TRECD* <FIELDLIST> 'END' *FINREC*

<COMPTYP>       <TYPEL>
```

```
                <IDENT> <SUB>
                <INCONST> *TSUB* 'PT2' <INCONST> *FINSUB*

<TYP>           <CLFITYP>
                'FILE' *TFICH* 'OF' <CLFITYP> *FINCLFI*
                'CLASS' *TCLASS* 'INTEGER' 'OF' <CLFITYP> *FINCLFI*

<TYPDF>         <TYPEL>
                'FILE' *TFICH* 'OF' <CLFITYP>
                'CLASS' *TCLASS* 'INTEGER' 'OF' <CLFITYP> *FINCLFI*
                <CONSTANT> *TSUB* 'PT2' <CONSTANT> *FINSUB*

<CONSTINT>      &RCS& <IDENT>
                <SIGN> &RCS& 'INTEGER'
                'QUOT' &RCS& 'CHAR' 'QUOT'

<SUB>           *VALID* 'PT2' <CONSTINT> *FINSUB*
                *IDTYP*

<BASTYP>        <IDENT> <SUB>
                <SIGN> *TENT* 'INTEGER' 'PT2' <CONSTINT> *FINSUB*
                'QUOT' *TCAR* 'CHAR' 'QUOT' 'PT2' <CONSTINT> *FINSUB*

<INDTYPLIST>    <INDTYP> <S23>

<S23>           *NINDEX* 'VG' <INDTYP> <S23>
                *FINDEX*

<INDTYP>        <BASTYP>
                *TSCAL* 'PAR1' <IDLIST> 'PAR2' *FINSCAL*

<SIGN>          'PLUS'
                &NEGA& 'MOINS'
                ()

<PARAMLIST>     <PARAM> <S10>

<S10>           'PVG' <PARAMLIST>
                ()

<PARAM>         *MPASS* <PREFIXE> <IDLISP> 'DPT' <IDENT> *TPARAM*
                *PPROC* 'PROCEDURE' <IDLISP>

<PREFIXE>       'VALUE'
                'VAR'
                'FUNCTION'
                ()

<IDLISP>        *IDPARAM* <IDENL> <S41>

<S41>           'VG' <IDLISP>
                ()

<CHARLIST>      &LCHAIN& 'CHAR' <CHARLIST>
                ()

<IDENL>         *IDLOC* 'IDENG'
```

```
<IDENT>         &RECOGNIZE 'IDENT'

<NUMBER>        'INTEGER'
                'REAL'

<EXP>           <SIMPLEXP> <RELEXP>

<RELEXP>        &PUSHUPB <RELOP> <SIMPLEXP> &POLLOPB
                ()

<SIMPLEXP>      <SIGN> <TERM> <ADDEXP>

<ADDEXP>        &PUSHUPB <ADDOP> <TERM> <ADDEXP> &POLLOPB
                ()

<TERM>          <FACTOR> &POLLMULB <MULTEXP>

<MULTEXP>       &PUSHUPB <MULTOP> <FACTOR> <MULTEXP> &POLLOPB
                ()

<ADDOP>         'PLUS'
                'MINUS'
                'OR'

<MULTOP>        'MUL'
                'DIVI'
                'DIV'
                'MOD'
                'ET'

<RELOP>         'EQ'
                'NEQ'
                'SUP'
                'INF'
                'SUPEQ'
                'INFEQ'
                'IN'

<FACTOR>        &VALNIL 'NIL'
                <CONSTEL>
                <IDENT> &VARFUNC <POSTFIXE>
                'PAR1' <EXP> 'PAR2'
                'NOT' <FACTOR> &POLLNEG
                &INITSET 'CRE1' <EXPVIR> &GENSET 'CRE2'

<POSTFIXE>      &APPEL 'PAR1' <ACTLIST> 'PAR2' &GENAPP
                &VARSIMP
                <ENTVAR> <COMPVAR> &VARSTRUC

<EXPVIR>        &CENTBIT <EXP> <S24>
                ()

<S24>           'VG' &CENTBIT <EXP> <S24>
                ()

<EXPLIST>       <EXP> &INILEX <S15>

<S15>           'VG' <EXP> &INCRDEX <S15>
                ()
```

```
<FIELDLIST>     <COMPART>
                <VARPART>

<COMPART>       <IDENLIST> 'LPT' <CLFITYP> *TCHAMP* <S14>

<S14>           'PVG' <FIELDLIST>
                ()

<VARPART>       'CASE' <IDENT> *TAG* 'OPT' <IDENT> *TTAG* 'OF' <VARFIELD>

<VARFIELD>      <CONSTANTLIST> 'OPT' <VARIANT> <S15>

<S15>           'PVG' <VARFIELD>
                *FINTAG*

<VARIANT>       *BRANCH* 'PAR1' <FIELDLIST> 'PAR2' *FINBR*
                ()

<IDENLIST>      <IDENT> *CTVAR* <S44>

<S44>           'VG' <IDENLIST>
                ()

<VARIABLE>      <IDENT> &VARFENC& <COMPVAR>

<COMPVAR>       <ENTVAR> <COMPVAR>
                ()

<ENTVAR>        'CRO1' <EXPLIST> 'CRO2' &GENIND&
                'PT' <IDENT> &GENCHAM&
                'FLECHE' &GENPOINT&

<STATEMENT>     <IDENT> &VARFENC& <ASCALL>
                'BEGIN' <TAIL> 'END'
                'CASE' <EXP> &GENCAS& 'OF' <CASTAT> 'END' &MISADD&
                'IF' <EXP> &GENIF& 'THEN' <STATEMENT> <ELSTAT> &FINCAS&
                &GENBOUC& 'WHILE' <EXP> &GENDO& 'DO' <STATEMENT> &GENCYC&
                &GENBOUC& 'LOOP' <TAIL> 'EXIT' 'IF' <EXP> &GENSAUT& 'PVG' <TAIL>
                &GENBOUC& 'REPEAT' <TAIL> 'UNTIL' <EXP> &GENUNT&    'END' &GENCYC&
                'WITH' <VARIABLIST> &GENWT& 'DO' <STATEMENT> &FINWT&
                'FOR' <IDENT> &RIEN 'AFF' <EXP> &RELIM& <LIMIT> <EXP> &GENFOR& 'DO'
                'GOTO' <GOTOSTAT>                          <STATEMENT> &GENTOU&
                ()

<ASCALL>        <ENTVAR> <COMPVAR> 'AFF' <EXP> &ACCES&
                'AFF' <EXP> &ACCESIM&
                &APPEL& 'PAR1' <ACTLIST> 'PAR2' &GENAPP&
                &GENAPSP&

<ACTLIST>       <EXP> &NACTUEL& <S25>

<S25>           'VG' <ACTLIST>
                ()

<ELSTAT>        &GENELSE& 'ELSE' <STATEMENT>
                *FICE* <PARELSE>
```

```
<PARALLEL>          'PVC'
                    'UNTIL'
                    'END'
                    'EXIT'

<LIMIT>             'TO'
                    'DOWNTO'

<VARIABLELIST>      <VARIABLE> &VLIST& <S17>

<S17>               'VG' &NVLIST& <VARIABLIST>
                    ()

<CASTAT>            <CONSTANTLIST> &GENDEF& 'OPT' <STATEMENT> &FINCASE& <S16>

<S16>               'PVG' <CASTAT>
                    ()

<CONSTANTLIST>
                    <CONSTANT> &LABCASE& <S19>

<S17>               'VG' <CONSTANTLIST>
                    ()

<DEFLSTAT>          &GENDEB 'INTEGER'
                    'EXIT' &GENEXG 'INTEGER'
```

Légende:

<~~>	Symbole non terminal
'~~'	Symbole terminal
~~	Fonction du module SEMANT
&~~&	Fonction du module MACRO
()	Alternative vide

ANNEXE III

DESCRIPTEURS

VARIABLES

descripteurs	préfixe TD TB I	accès au type PT(11 bits)	accès à la valeur PV(16 bits)
SCALAIRE	01 00 I	Adresse Type	Valeur
ENTIER COURT	01 01 I	"	Valeur
POINTEUR	01 10 I	"	"
BOOLEEN	01 11 I		Valeur
ENTIER	00 0i	VALEUR	
REEL	00 1i	VALEUR	
TABLEAU,RECORD	10 10 0	Adresse Type	Adresse Valeur
CLASSE,FICHIER	10 11 M	"	"
INTERVALLE	10 00 0	"	"
POWERSET	10 01 0	"	"
REF. DESC.	10 00 1	"	"
ETIQ.EXTERNE	10 01 1	Déplacement	Adresse début segment
LITT. CHAINE	10 10 1	Longueur	Adresse valeur (CODE)

VALEURS

type		longueur en bits 1	7	8	16
SCALAIRE		i			
BOOLEEN		i			
POINTEUR		i	Valeur		
POWERSET		i		VALEUR	
REEL		V	01 i	"	
ENTIER		V	00 i	"	
ENTIER COURT		i	Valeur		

Signification des lettres du Préfixe

- I — Bit d'indirection
- i — Bit d'initialisation
- M — Mode du fichier (Entrée/Sortie)
- V — Taille non standard de la valeur

TYPES

descripteurs	préfixe PF	accès au type PT	taille PV T1 / T2
SCALAIRE	1 1 100	maximum	Adresse méthode codage
SOUS-SCALAIRE	1 1 101	Ad.Type base	MIN / MAX.
POINTEUR	1 0 tvl	Ad.Type élém.	
CLASSE	1 1 001	"	Taille max.
TABLEAU	0 0 tvl	"	Taille
"DOPE-VECTOR"	0 1 00D	Borne inf.	Pas
RECORD	1 1 010	Ad.1° Branche	Taille max.
CHAMP	1 1 110	Ad. Type	Ad.valeur dans le Record
	T . tvl	Ad.Branche	
BRANCHE	1 1 011	Ad.Tagfield	Corr.Taille / 1°Etiquette
	F . Net	Ad.Frère	(Ad.Fils)
FICHIER	1 1 000	Ad.Type élem.	Ad. Desc. physique
	S . tvl	(nombre d'élem.)	
SOUS-REEL	0 1 11	MINIMUM	
		MAXIMUM	
SOUS-ENTIER	0 1 10	MINIMUM	
		MAXIMUM	
SOUS-ENTIER-COURT	0 1 01	Minimum	Maximum

Signification des lettres du Préfixe

- F — Labranche possède un fils
- T — Le champ est un "tagfield"
- D — Dernier élément du "dope-vector"
- S — Fichier séquentiel ou direct
- Net — Nombre étiquettes
- tvl — Type valeur

  000 entier
  001 réel
  010 powerset
  011 entier-court
  100 scalaire
  101 structure
  110 pointeur
  111 booléen

CONTROLE

DESCRIPTEURS	PREFIXE	PT	PV T1 T2
PROCEDURE OU	1111 O	Nb. var. local	Adresse code
FONCTION	Nive P	Ad. Type	Nb. Param
PARAM. FORMEL	1110 R	Ad. Type	
BOUCLE	1101 O		Adresse sébut Segment
WITH	1100 O	With préc.	Adresse début Segment
CASE IF IFELSE	1100 1		Adresse fin Segment
FOR	1101 O		Adresse début Segment
	PF U	Valeur de la limite	
SEG. PRINC.	1101 1		Adresse Retour
MARQUE DE BLOC	1111 1	Niveau absolu	Chaînage dynamique
	P	Ad.Segm.With	Chaînage statique

Signification des lettres du préfixe

- P Procédure ou fonction
- U "TO" ou "DOWNTO"
- R Mode de passage (Référence/Valeur)
- Nive Niveau lexicographique

INSTRUCTIONS

INSTRUCTIONS	1° octet	2° octet	3° octet	4° octet
NOM S,D	0 1 0 0 S	D		
ID S,D	0 1 0 1 S	D		
AFFECT S,D	0 1 1 0 S	D		
TO S,D	0 1 1 1 S	D		
IF m	0 0 1 0	m		
CASE m	0 0 1 1	m		
DO m	0 0 0 1	m		
INDEX n	1 0 1 0 0	n		
WITH n	1 0 1 0 1	n		
UNZER P,Φ	1 0 1 1 0	P|Φ		
CHAMP p	1 0 0 0 0	p		
CHAMP i,j;p	1 0 0 1 1	p	i j	
CREER n,P,p	1 0 0 1 0	p	n p	
LITT P,V	1 0 0 0 1	P V		
APPEL S,S;n	1 1 0 0 0 0 0 0	S D	n	
INDEX S,D;n	1 1 0 0 0 0 0 1	S D	n	
ALLOC S,D;n	1 1 0 0 0 0 1 0	S D	n	
RESET S,D	1 1 0 0 0 0 1 1	S D		
EXIT S,D	1 1 0 0 0 1 0 0	S D		
GO n,m	1 1 0 0 0 1 1 0	m	n	
OF n,m	1 1 0 0 0 1 0 1	m	n	
FORUP S,D;n	1 1 0 0 1 0 0 0	S D	m	
FORDOWN S,D;m	1 1 0 0 1 0 0 1	S D	m	
IFELSE m1,m2	1 1 0 0 1 0 1 0	m1	m2	
CREER n,P	1 1 0 0 1 1 0 0	n P		
BIT n	1 1 0 0 1 1 0 1	n		

Toutes les autres instructions ont un code opération de 8 bits aussi et pas d'opérandes (d'où une longueur totale d'un octet).

Remarque: Le code opération est indiqué ici en binaire et les opérandes en symbolique.

ANNEXE IV

Représentation dans le Langage intermédiaire des instructions PASCAL

1/ Affectation et expression

*A(I,J).U:=X↑.V+1-B(4-Y) ;

	NOM S_X,D_X
	POINT 0
	CHAMP P_V
	UNZER 'E',1
expression	LITT 'E',4
	NOM S_y,D_y
	MOIN
	INDEX S_b,D_b;1
	MOIN
	PLUS
	NOM S_i,D_i
variable	FINEXP
	NOM S_j,D_j
	INDEX S_a,D_a;2
affectation	CHAMP P_u
	AFFECT

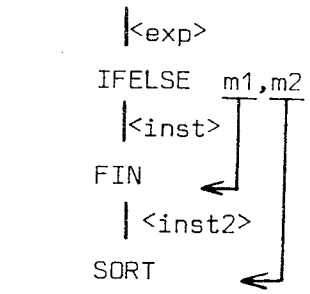
*B:=A|[bleu,succ(I)] &C ;

	LITT 'S','num(bleu)'
	FINEXP
	NOM S_i,D_i
expression	SUCC
	BIT 2
	NOM S_c,D_c
	ET
	NOM S_a,D_a
	OU
	AFFECT S_b,D_b

2/ if <exp> then <inst>;

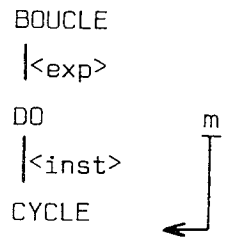
	|<exp>
	IF
	|<inst>
	SORT

if <exp> then <inst> else <inst2> ;

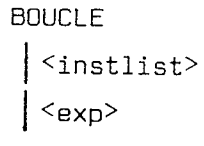


3/ Boucles

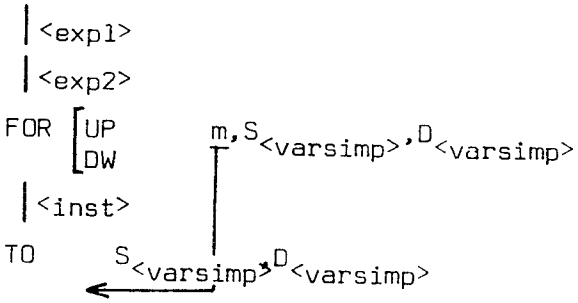
while <exp> do <inst> ;



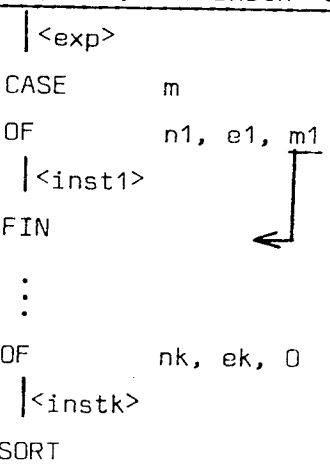
repeat <instlist> until <exp> ;



for <varsimp>:= <expl> to <exp2> do <inst> ;



4/ case <exp> of e1:<inst1>;....; ek:<instk> end ;



5 - with <v_1>, ...,<v_n> do <$inst_1$>; B:=5; <inst2> end ;

avec B champ de la variable v_i

```
  |<v1>
FINEXP
  :
  |<vn>
WITH        n
  |<inst1>
LITT        'E',5
CHAMW       0,i,P
AFFECT
  |<inst2>
SORT
```

6 - appel de procédure ou fonction

PF (<exp1>, ...,<expn>);

```
| <exp 1>
FINEXP
  :
| <expn>
APPEL Spf,Dpf;n
```

7 - déclarations

A, B, C: integer ;

```
CREEST  3,'int'
```

M: array (1..20) of char;

```
CREER   1,'tab', <adresse type>
```

ANNEXE V

```
//PASCAL JOB
/*MAIN LINES=(5,C)
//COMPIL EXEC PGM=PASCAL
//STEPLIB DD DSN=S5017.P0340.PASCAL.CLCAOKOK,DISP=SHR
//SYSUDUMP DD DUMMY
//OUTPUT DD SYSOUT=A,DCB=(RECFM=F,BLKSIZE=133,LRECL=133)
//SYSLIN DD DSN=&SYSL,UNIT=PDISK,DISP=(NEW,PASS),SPACE=(TRK,(10,5)),
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=1600)
//PMDFILE DD DSN=&PMD,UNIT=PDISK,DISP=(NEW,DELETE),SPACE=(TRK,(1,1)),
//    DCB=(RECFM=F,BLKSIZE=20,LRECL=20)
//CHEFILE DD DSN=&CHF,DISP=(NEW,DELETE),UNIT=PDISK,SPACE=(TRK,(1,1)),
//    DCB=(RECFM=FB,LRECL=1,BLKSIZE=256)
//INPUT DD *
            /* INTERPRETEUR PASCAL */
            /*      */
            /* VERSION FORTIER DU 15-9-74 */
            /*      */
 LABEL 100 ;
            /*      */
 CONST
          PREDEF=0,
          GLOBAL=1,
          LOCAL=2,
          INT2=3,          /* TYPE NL */
          INT3=4,
          INT4=5,
          INT5=6,
       /*      */          /* TYPE NLA */
          S0=0,S1=1,S2=2,S3=3,S4=4,S5=5,S6=6,
       /*      */
      /*   CODE    MNEMO ASM                          CODE    MNEMO ASM   */
NOM       =  9, /* NAME      */    INN       =  4, /* INE     */
ID        = 31, /* NAMIND    */    BIT       =  7, /* BIT     */
VI        = 20, /* VIM       */    ALOC      =  6, /* ALOB    */
DESCRIM   = 58, /* DESCKRIM  */    RES       = 18, /* RES     */
LITT      = 37, /* LITT      */    SUC       = 47, /* SUIV    */
UNZER     = 60, /* UNZER     */    PRE       = 15, /* PRE     */
                                   ABSOLU    = 23, /* ASOL    */
CREE      = 25, /* CREER     */    TRONK     = 54, /* TRONK   */
CREP      = 36, /* CREPR     */    MODULO    = 56, /* MODULO  */
CREST     = 41, /* CREST     */
                                   DIVENT    = 34, /* DIVIN   */
CHAMP     = 17, /* CHAMP     */    DIVIZ     = 39, /* DIVIS   */
IND       = 21, /* INDR      */    MULT      = 42, /* MULT    */
CHAMW     = 24, /* CHAMW     */    PLUS      = 44, /* PLUS    */
INDEX     = 27, /* INDX      */    MOIN      = 46, /* MOINS   */
POINT     = 50, /* POINT     */
                                   AFF       = 45, /* ASSIGN  */
DIFFER    =  0, /* DIFF      */    AFFC      = 43, /* ASCOMP  */
EG        = 11, /* EGAL      */
NEGA      =  3, /* NEGA      */    EXEC      =  8, /* EXC     */
NON       = 19, /* NON       */    APPEL     = 26, /* APPEL   */
ET        =  1, /* ET        */    RETOUR    = 14, /* RT      */
OU        = 12, /* OU        */
INF       =  5, /* INF       */    IFS       = 10, /* IFS     */
```

```
INFEG     = 30, /* INFEGAL   */          IFELSE    = 22, /* IFLS    */
SUP       = 55, /* SUPER     */
SUPEG     = 57, /* SUPEGAL   */          STOP      = 51, /* STOPE   */
FORUP     = 13, /* UP        */          TOU       = 32, /* TOU     */
FORBOT    = 33, /* BOTT      */          WT        = 28, /* WIT     */
KASE      = 35, /* CHOIX     */          UNTI      = 59, /* UNTYC   */
OFS       = 16, /* OFS       */          GO        = 29, /* ALEZI   */
BOU       = 40, /* BOUKO     */          EXIT      = 38, /* EXITD   */
BOUC      = 53, /* BOUKKLE   */          FIN       =  2, /* FIK     */
CYC       = 52, /* SYKLI     */          SORT      = 61, /* ASORTIE*/
FSPE      = 49, /* FSPECX    */
      /*      */
   PGET   = 0,      FEOF  = 4,
   PPUT   = 1,      FINT  =5,
   PREAD  = 2,      FCHR  =6,
   PWRITE = 3,      FODD  =7,
         /*            */
         BOOL=0,
         SCAL=1,
         ENTC=2,          /* TYPE OVAL */
         REF=3,
         ENT=4,
         REEL=5,
         POW=6,
         CH=7,

         SUBEC=0,         TPSCAL=9,
         CLAS=1,          SUBENT=10,
         REC=2,           SUBREL=11,      /* TYPE OTYP */
         SUBC=3,          CHAM=12,
         OV=4,            TAG=13,
         OVD=5,           BRANCH=14,
         POIN=6,          ETI=15,
         FICH=7,
         TAB=8,
      /*      */
         DENT=0,          ENTCIND=9,      LAB=18,
         SUBE=1,          OREF=10,        PROC=19,
         OREEL=2,         REFIND=11,      FONC=20,
         SUBR=3,          STSTMP=12,      PARM=21,
         OBOOL=4,         LCH=13,         MARKP=22,
         BOOLIND=5,       OPOW=14,        MARKF=23,
         OSCAL=6,         STCOMP=15,      SEGP=24,
         SCALIND=7,       VALEUR=16,      BOUK=25,
         DENTC=8,         RDESC=17,       FORTO=26,
         FORDOWN=27,      ALT=28,         WTH=29,
         OCH =30,         CHIND =31,
      /*      */
   PR=0,
   FO=1,
   REFER=2,
   VALU=3,
         /*       */
    NTYP=50,
    NINST=500, /* CONSTANTES */
```

```
    NDESC=1000,
    NT=20,
    VALMAX=30 ;
          /*      */
TYPE
    TSCAL=0..255;
    TPOW=POWERSET 0..VALMAX ;
       /*      */
    SHRT=-077777B..077777B ;
       /*      */
         NL=0..6 ;
         NLA=0..6 ;
         CODOP=0..61 ;
         OVAL=0..7 ;
         OTYP=0..15 ;
         TDESC=0..31 ;
         MODEPARM=0..3;
          /*      */
TVAL=
    RECORD
       NI : BOOLEAN ; /* BIT DE NON-INITIALISATION */
       CASE PFV : OVAL OF
       BOOL : (VB : BOOLEAN) ;
       SCAL : (VS : TSCAL) ;
       ENTC : (VEC : SHRT) ;
       REF : (VAD : SHRT) ;
       ENT : (VE : INTEGER) ;
       REEL : (VR : REAL) ;
       POW : (VP : TPOW) ;
       CH : (VCH : CHAR)
    END ;
          /*      */
DESTP=
    RECORD
       CASE PFT : OTYP OF
       SUBEC /*,CLAS,REC,SUBC,DV,DVD*/ :
          (TE,TAIL,BA : SHRT) ;
       POIN /*,FICH,TAB*/ :
          (PFEL : TDESC ; TEL,TM : SHRT) ;
       TPSCAL :
          (TB : SHRT ; MI,MA : TSCAL) ;
       SUBENT :
          (MIE,MAE : INTEGER) ;
       SUBREL :
          (MIR,MAR : REAL) ;
       CHAM /*,TAG*/ :
          (PFC : TDESC ; TP,ADRL,BR : SHRT) ;
       BRANCH :
          (PTAG,FRER,FILS,CORT,NETI : SHRT) ;
       ETI :
          (ETIQ : ARRAY(0..4) OF TSCAL)
    END ;
          /*      */
DESC=
    RECORD
```

```
      . CASE PF : TDESC OF
        DENT /*,DREEL,DCH*/ :
           (VALD : TVAL) ;
        DBOOL /*,DSCAL,DENTC,DREF*/ :
           (PTS : SHRT ; DVAL : TVAL) ;
        VALEUR :
           (VAL : TVAL) ;
        BOOLIND /*,SCALIND,ENTCIND,REFIND,CHIND,
        LAB,STSIMP,STCOMP,SURF,SUBR,
        DPOW,RDESC,LCH*/ :
           (PT,PV : SHRT) ;
        PROC /*,FONC*/ :
           (NIV : NLA ; PFF : TDESC ;
           ADC,NP,TF,NVL : SHRT) ;
        PARM :
           (MODE : MODEPARM ;
            PFP : TDESC ; TP : SHRT) ;
        MARKP /*,MARKF*/ :
           (NV : NLA ; SEGCT,STA,DYN : SHRT) ;
        SEGP :
           (ADRET : SHRT) ;
        ALT /*,BOUK,FORTO,FORDOWN*/ :
           (ADS1 : SHRT) ;
        WTH :
           (ADS2,AFS2 : SHRT)
    END ;
           /*     */
INST=
    RECORD
       CASE OP : CODOP OF
       CREP /*,IND,CHAMP,WT,BIT,BOO,KASE,
       FORUP,FORBOT,IFS,FSPE*/ :
          (L : SHRT) ;
       NOM /*,ID,AFF,TOU,EX,GC,RES*/ :
          (S1 : NL ; D1 : SHRT) ;
       CREST /*,LITT,UNZER*/ :
          (P : TDESC ; V : TVAL) ;
       IFELSE /*,OFS*/ :
          (M1,M2 : SHRT) ;
       CREE :
          (PX : TDESC ; TP,NB : SHRT) ;
       APPEL /*,CHAMW,INDEX,ALOC*/ :
          (S2 : NL ; D2,N : SHRT) ;
       VI : (VAL : TVAL);
       DESCIM : (DESCRIM : DESC)
    END ;
           /*     */
           /* VARIABLES */
           /*     */
VAR
           /*     */
    DVAR,DP : DESC ; /* TRAVAIL SUR DESC */
    TCT : SHRT ; /* INDEX DANS DTYP */
    CTRL : DESC ; /* DESC DE CONTROLE */
    EVALCT,L : SHRT ; /* INDEX DANS CACHEVAL */
```

```
    GLOB,LOC,SEG,LIBRE : SHRT ; /* INDEX DANS CTX */
    RI : INST ; /* REGISTRE INSTRUCTION */
    CO : SHRT ; /* COMPTEUR ORDINAL */
    T : SHRT ; /* ADRESSE TYPE */
    AD : SHRT;     /* TRAVAIL SUR LES ADRESSES */
    I,J : SHRT ;
    K : BOOLEAN ;
    ARRET : BOOLEAN ;
    INDIR,DIRECT : POWERSET TDESC ;
    TRAV1 : TVAL ;
    TRAV,NB : SHRT ;
    FICHIER,OUTPUT : FILE OF ARRAY(0..132) OF CHAR ;
    INPUT : FILE OF ARRAY(1..80) OF CHAR ;
    TEXTE : FILE OF INST ;
     /*       */
    TYP : ARRAY(0..NTYP) OF DESTP ;
    PROG : ARRAY(0..NINST) OF INST ;
    CACHEVAL : ARRAY(0..NT) OF DESC ;
    DTYP : ARRAY(1..4) OF DESTP ;
    CTX : ARRAY(-10..NDESC) OF DESC ;


 VALUE
    TCT=0 ;
    EVALCT=0 ;
    GLOB=0 ;
    LOC=0 ;
    SEG=0 ;
          /*      */
       /* PROCEDURES */
          /*      */
PROCEDURE ERREUR (N : SHRT); FORWARD;
         /*      */
PROCEDURE CHARGE;
 BEGIN
  CO:=0;
  WHILE TEXTE".OP<>STOP DO
  BEGIN
   PROG(CO):=TEXTE";
   CO:=SUCC(CO);
   IF CO > NINST THEN ERREUR(21);
   GET(TEXTE)
  END;
  PROG(CO):=TEXTE";
 END; /*CHARGE */


       /*  PROCEDURES UTILITAIRES */


PROCEDURE WRITESHRT(NB,POS:SHRT);
 VAR I :SHRT; NEG:BOOLEAN;
 BEGIN
  I:=POS;
```

```
  N:=NB;
  IF N<0 THEN
  BEGIN
    N:=-N;
   NEG:=TRUE
  END ELSE  NEG:=FALSE;
  REPEAT
    FICHIER"(I):=CHR(MOD(N,10)+INT('0'));
    N:=N DIV 10;
    I :=PRED(I)
  UNTIL  N:=0;
  IF NEG THEN FICHIER"(I):='-'
END;
FUNCTION CALCAD(S : NL ; D : SHRT) : SHRT ;
    VAR I : SHRT ;
    BEGIN
       CASE S OF
    PREDEF : CALCAD:=GLOB-D ;
    GLOBAL : CALCAD:=GLOB+D ;
    LOCAL : CALCAD:=LOC+D ;
    INT2,INT3,INT4,INT5 :
       BEGIN DVAR:=CTX(LOC) ;
           I:=DVAR.DYN ;
           DVAR:=CTX(I) ;
           WHILE INT(DVAR.NV) >= INT(PRED(S)) DO
           BEGIN
           I:=DVAR.STA ;
           DVAR:=CTX(I)
           END ;
           CALCAD:=I+D
       END
    END    /*  CASE */
    END ; /* CALCAD */
           /*      */
FUNCTION ADSPE(I : NL ; J : SHRT) : SHRT ;
    VAR K,L : SHRT ;
    BEGIN
       K:=SEG ;
       L:=INT(I) ;
       DP:=CTX(K) ;
       WHILE L >= 0 DO
       BEGIN
          K:=CTRL.AFS2 ;
          DP:=CTX(K) ;
          L:=L-1
       END ;
       ADSPE:=K+J
    END ; /* ADSPE */
           /*      */
PROCEDURE TEST ;
    BEGIN
       IF ¬ (DVAR.PF IN[<DBOOL,BOOLIND>])
       THEN ERREUR(11) ;
       IF DVAR.PF=BOOLIND
       THEN DVAR.DVAL:=CTX(DVAR.PV).VAL ;
```

```
      IF DVAR.DVAL.NI
      THEN ERREUR(6)
   END ; /* TEST */
         /*     */
PROCEDURE PUSHVAL(X : DESC) ;
   BEGIN
      CACHEVAL(EVALCT):=X ;
      EVALCT:=SUCC(EVALCT) ;
      IF EVALCT > NT THEN ERREUR(17)
   END ; /* PUSHVAL */
         /*     */
PROCEDURE POPVAL(VAR X : DESC) ;
   BEGIN
      EVALCT:=PRED(EVALCT) ;
      IF EVALCT < 0 THEN ERREUR(18) ;
      X:=CACHEVAL(EVALCT)
   END ; /* POPVAL */
         /*     */
PROCEDURE DESCEND(N : SHRT) ;
   BEGIN
      EVALCT:=EVALCT-N ;
      IF EVALCT < 0 THEN ERREUR(18) ;
      L:=PRED(EVALCT)
   END ; /* DESCEND */
         /*     */
PROCEDURE MONTE(VAR J : SHRT) ;
   BEGIN
      J:=L ;
      L:=SUCC(L)
   END ; /* MONTE */
         /*     */
PROCEDURE PUSHTYP(X : SHRT) ;
   BEGIN
      TCT:=SUCC(TCT) ;
      IF TCT > 4 THEN ERREUR(19) ;
      DTYP(TCT):=TYP(X)
   END ; /* PUSHTYP */
         /*     */
PROCEDURE ERREUR ;
BEGIN
  FICHIER":='  ';
  PUT(FICHIER);
  FICHIER":='  CO=';
  WRITESHRT(CO,20);
  PUT(FICHIER);
   CASE N OF
      0 : FICHIER":='CHAMP INCOMPATIBLE AVEC TAGFIELD' ;
      1 : FICHIER":='PAS UN TABLEAU' ;
      2 : FICHIER":='DEPASSEMENT BORNES TABLEAU' ;
      3 : FICHIER":='EXIT INCORRECT' ;
      4 : FICHIER":='NOMBRE INDEXATIONS INCORRECT' ;
      5 : FICHIER":='VALEUR TROP GRANDE POUR UN POWERSET' ;
      6 : FICHIER":='VALEUR NON INITIALISEE' ;
      7 : FICHIER":='TYPES INCOMPATIBLES' ;
      8 : FICHIER":='VALEUR HORS DES BORNES' ;
```

```
      9 : FICHIER":='NOMBRE DE TAGFIELD INCORRECT' ;
      10 : FICHIER":='NOMBRE DE PARAM INCORRECT' ;
      11 : FICHIER":='EXP NON BOOLEENE DANS TEST' ;
      12 : FICHIER":='RECORD N''A PAS DE TAGFIELD' ;
      13 : FICHIER":='NOMBRE DE TAGFIELD INCORRECT DANS ALLOC' ;
      14 : FICHIER":='EXP N''EST PAS UN SCALAIRE SIMPLE' ;
      15 : FICHIER":='EXP NE FIGURE PAS DANS LA CASE' ;
      16 : FICHIER":='ERREUR PARAMETRE' ;
      17 : FICHIER":='CACHEVAL PLEIN' ;
      18 : FICHIER":='CACHEVAL VIDE' ;
      19 : FICHIER":='DTYP PLEIN' ;
      20 : FICHIER":='CTX PLEINE' ;
      21 : FICHIER":='PROGRAMME TROP GRAND'
   END ;
   PUT(FICHIER) ;
   IF N > 0 THEN
   GOTO EXIT100
END ; /* ERREUR */
         /*     */
PROCEDURE AFFPRIM(VAR X : TVAL) ;
   BEGIN
      WITH DP DO BEGIN
      IF PF IN|<DVAR.PF,SUCC(DVAR.PF)>| THEN
      BEGIN
         IF PF > DVAR.PF
         THEN DVAL:=CTX(PV).VAL ;
         IF DVAL.NI THEN ERREUR(6) ;
         X:=DVAL
      END ;
      IF (DVAR.PTS >= PTS) & (DVAR.PF¬=DBOOL) THEN
      BEGIN
         DTYP(1):=TYP(DVAR.PTS) ;


   /* ============================== I N T 1 - I N T 2 ================= */


         CASE DVAR.PF OF
         DSCAL :
         IF (X.VS < DTYP(1).MI) | (X.VS > DTYP(1).MA)
         THEN ERREUR(8) ;
         DENTC :
         IF (X.VEC < DTYP(1).TF) | (X.VEC > DTYP(1).TAIL)
         THEN ERREUR(8) ;
         DREF :
         ERREUR(8)
         END /* CASE */
      END
      ELSE
      BEGIN
         IF (DVAR.PF>=DENTC) & (¬(PF IN |<DENT,SUBE>|))
         THEN ERREUR(7) ;
         DTYP(1):=TYP(DVAR.PTS) ;
         IF PF=SUBE
         THEN VALD:=CTX(PV).VAL ;
```

```
        IF VALD.NI THEN ERREUR(6) ;
        IF (VALD.VE < DTYP(1).TE) | (VALD.VE > DTYP(1).TAIL)
        THEN ERREUR(8) ;
        X.VEC:=VALD.VE ;
        X.NI:=FALSE
      END /* ELSE */
      END /* WITH */
   END ; /* AFFPRIM */
        /*     */
PROCEDURE AFFRENT(VAR X : TVAL) ;
   BEGIN
      WITH DP DO
        IF PF IN|<DVAR.PF,SUCC(DVAR.PF)>| THEN
        BEGIN
         IF PF > DVAR.PF THEN VALD:=CTX(PV).VAL ;
         IF VALD.NI THEN ERREUR(6) ;
         X:=VALD
        END ELSE
        BEGIN
         IF PF IN|<DENTC,ENTCIND>| THEN
          BEGIN
           IF PF=ENTCIND THEN DVAL:=CTX(PV).VAL ;
           IF DVAL.NI THEN ERREUR(6) ;
           X.NI:=FALSE ;
           IF DVAR.PF=DENT THEN X.VE:=DVAL.VEC
           ELSE X.VR:=DVAL.VEC
          END ELSE
          BEGIN
           IF PF=SUBE THEN VALD:=CTX(PV).VAL ;
           IF VALD.NI THEN ERREUR(6) ;
           X.VR:=VALD.VE ;
           X.NI:=FALSE
          END
        END
   END ; /* AFFRENT */
        /*     */
PROCEDURE AFSUB(ADD : SHRT) ;
   BEGIN
      DVAR.PF:=PRED(DVAR.PF) ;
      AFFRENT(TRAV1) ;
      IF DVAR.PT <> 0 THEN
      BEGIN
        DTYP(1):=TYP(DVAR.PT) ;
        IF DP.PF = SUBE
        THEN IF (TRAV1.VE<DTYP(1).MIE)|(TRAV1.VE>DTYP(1).MAE)
         THEN ERREUR(8)
         ELSE
        ELSE IF (TRAV1.VR<DTYP(1).MIR)|(TRAV1.VR>DTYP(1).MAR)
         THEN ERREUR(8);
        CTX(ADD).VAL:=TRAV1
      END
   END ; /* AFSUB */
        /*     */
PROCEDURE AFSTRUC ;
   BEGIN
```

```
      IF ¬(DP.PF IN |< STSIMP,LCH >|) THEN ERREUR(7);
      DTYP(1):=TYP(DVAR.PT) ;
      IF DP.PF=LCH THEN
      BEGIN
         IF DTYP(1).PFEL¬=DCH THEN ERREUR(7);
         FOR I:=0 TO (DTYP(1).TM-1) DO
         IF I <= DP.PT THEN
         CTX(DVAR.PV+I).VAL := PROG(DP.PV+I).VAL
          ELSE BEGIN
           CTX(DVAR.PV+I).VAL.NI :=FALSE;
           CTX(DVAR.PV+I).VAL.VCH := ' '
          END
      END
   END ; /* AFSTRUC */
         /*     */
PROCEDURE AFBOUC ;
   BEGIN
      CASE DVAR.PF OF
        DENT,DCH :  CTX(AD) := DP ;
        DSCAL : BEGIN
         DVAR.DVAL.NI:=FALSE ;
         DVAR.DVAL.VS:= J  ;
         CTX(AD):=DVAR
         END ;
        DENTC : BEGIN
         DVAR.DVAL.NI:=FALSE ;
         DVAR.DVAL.VEC:=J ;
         CTX(AD):=DVAR
         END ;
        SUBE : CTX(DVAR.PV).VAL:=DP.VALD ;
        SCALIND : BEGIN
         CTX(DVAR.PV).VAL.VS :=  J  ;
         CTX(DVAR.PV).VAL.NI:=FALSE
         END ;
        CHIND : BEGIN
         CTX(DVAR.PV).VAL.VCH := CHR(J);
         CTX(DVAR.PV).VAL.NI := FALSE
         END;
        ENTCIND : BEGIN
         CTX(DVAR.PV).VAL.VEC:=J ;
         CTX(DVAR.PV).VAL.NI:=FALSE
         END
      END /* CASE */
   END ; /* AFBOUC */
         /*     */
PROCEDURE ACCEPOW ;
   BEGIN
      IF DP.PV=0 THEN
      BEGIN
         POPVAL(DVAR) ;
         TRAV1:=DVAR.VAL
      END ELSE
       TRAV1 := CTX(DP.PV).VAL ;
      IF ((DVAR.PV ¬= 0) & (DP.PV ¬= 0) & (DVAR.PT ¬= DP.PT)) THEN
      ERREUR(7)   ;
```

```
        IF DVAR.PV=0 THEN POPVAL(DVAR)
        ELSE DVAR:=CTX(DVAR,PV) ;
        IF DVAR.VAL.NI | TRAV1.NI THEN ERREUR(6)
    END ; /* ACCEPOW */
        /*      */
PROCEDURE AFPOW(X : SHRT) ;
    BEGIN
        IF DP.PF ¬= DPOW THEN ERREUR(7) ;
        DTYP(1):=TYP(DVAR,PT) ;
        IF DP.PV ¬= 0 THEN
        BEGIN
            IF DP.PT ¬= DVAR.PT THEN ERREUR(7) ;
            CTX(X):=CTX(DP,PV)
        END ELSE
        BEGIN
            POPVAL(DP) ;
            CTX(X):=DP
        END
    END ; /* AFPOW */
        /*      */
PROCEDURE CONVENT ; /* CONVERSION EN ENTIER */
    BEGIN
        IF DVAR.PF IN (DIRECT | INDIR - |<CREEL,SUBR>|) THEN
        ERREUR(14) ;
        WITH DVAR DO
        CASE PF OF
          DENT : BEGIN
           11 : IF VALD.NI THEN ERREUR(6) ;
           J:=VALD.VE
           END ;
          DBOOL : BEGIN
           12 : IF DVAL.NI THEN ERREUR(6) ;
           J:=INT(DVAL.VB)
           END ;
          DSCAL : BEGIN
           13 : IF DVAL.NI THEN ERREUR(6) ;
           J:=INT(DVAL.VS)
           END ;
          DENTC : BEGIN
           14 : IF DVAL.NI THEN ERREUR(6) ;
           J:=DVAL.VEC
           END ;
          SUBE : BEGIN
           VALD:=CTX(PV).VAL ;
           GOTO 11
           END ;
          BOOLIND : BEGIN
           DVAL:=CTX(PV).VAL ;
           GOTO 12
           END ;
          SCALIND : BEGIN
           DVAL:=CTX(PV).VAL ;
           GOTO 13
           END ;
          ENTCIND : BEGIN
```

```
            GOTO 14
            END
         END /* CASE */
      END ; /* CONVENT */


PROCEDURE CONVENTRE ;
   BEGIN
     IF DVAR.PF < DENTC THEN
     BEGIN
        IF DVAR.PF IN  |<SUBE,SUBR>| THEN
        BEGIN
           DVAR.VALD:=CTX(DVAR.PV).VAL ;
           DVAR.PF:=PRED(DVAR.PF) /* SUBE/SUBR => DENT/DREEL */
        END ;
        IF DVAR.VALD.NI THEN ERREUR(6)
     END ELSE
     BEGIN
        IF DVAR.PF=ENTCIND THEN
        DVAR.DVAL:=CTX(DVAR.PV).VAL ;
        DVAR.VALD.VE:=DVAR.VALD.VEC ; /* ENTC => ENTIER */
        DVAR.PF:=DENT
     END
 END;  /* CONVENTRE */

            /*      */
PROCEDURE TESTAG; FORWARD;
            /*     */
 PROCEDURE CONTROL ;
     BEGIN
        REPEAT
           PUSHTYP(DTYP(2).BR) ; /* DESCRIPTEUR DE BRANCHES */
           DVAR:=CTX(DTYP(3).ADRL+AD) ;
           TESTAG ;
           IF ¬ K THEN ERREUR(0)
           ELSE DTYP(2):=DTYP(3) ; /* TEST TAGFIELD */
        UNTIL (DTYP(2).BR=0) | (¬ K)
     END ; /* CONTROL */
            /*      */
 PROCEDURE TESTAG ;
     BEGIN
     WITH DTYP(TOT) DO
     BEGIN
        K:=FALSE ;
        I:=NETI ;
        CONVENT ;
        TRAV1.VS:= J  ;
        FOR J:=1 TO I DO
        IF ETIQ(J)=TRAV1.VS THEN
        K:=TRUE
     END
     END ; /* TESTAG */
       /*      */
PROCEDURE PULLCTXL( VAR X:DESC);
BEGIN
```

```
 LIBRE := PRED(LIBRE) ;
 X := CTX(LIBRE)
END ;
        /*     */
PROCEDURE PUSHCTXL( X:DESC);
BEGIN
 LIBRE := SUCC(LIBRE) ;
 IF LIBRE >   NDESC THEN ERREUR(20) ;
 CTX(LIBRE) := X
 END ;
          /*        */
PROCEDURE PUSHCTXT( X:DESC);
 BEGIN   CTX(TRAV) := X ;
  TRAV := SUCC(TRAV)
END ;
          /*        */
        /*        */

     /*        */
FUNCTION Z(X,Y:TVAL ; OP:CODOP) : BOOLEAN;
BEGIN
 IF X.PFV¬=Y.PFV THEN ERREUR(99) ;      /*       CODE A AJOUTER  */
 CASE X.PFV OF
 BOOL : CASE OP OF
         EG : Z := X.VB=Y.VB;
         DIFFER : Z := X.VB¬=Y.VB;
         SUPEG : Z := X.VB>=Y.VB
         END;
 SCAL : CASE OP OF
         EG : Z := X.VS=Y.VS;
         DIFFER : Z := X.VS¬=Y.VS;
         SUPEG : Z := X.VS>=Y.VS
         END;
 ENTC : CASE OP OF
         EG : Z := X.VEC=Y.VEC;
         DIFFER : Z := X.VEC¬=Y.VEC;
         SUPEG : Z := X.VEC>=Y.VEC
         END;
 REF : CASE OP OF
        EG : Z := X.VAD=Y.VAD;
        DIFFER : Z := X.VAD¬=Y.VAD;
        SUPEG : Z := X.VAD>=Y.VAD
        END;
 ENT : CASE OP OF
        EG : Z := X.VE=Y.VE;
        DIFFER : Z := X.VE¬=Y.VE;
        SUPEG : Z := X.VE>=Y.VE
        END;
 REEL : CASE OP OF
         EG : Z := X.VR=Y.VR;
         DIFFER : Z := X.VR¬=Y.VR;
         SUPEG : Z := X.VR>=Y.VR
         END;
 POW : CASE OP OF
        EG : Z := X.VP=Y.VP;
```

```
        DIFFER : Z := X.VP¬=Y.VP;
        SUPEG : Z := X.VP>=Y.VP
        END;
 CH: CASE OP OF
      EG : Z := X.VCH = Y.VCH;
      DIFFER : Z:= X.VCH ¬= Y.VCH;
      SUPEG : Z := X.VCH >= Y.VCH
      END
 END
END;     /*   FUNCTION Z   */
   /*       */
   /*     */


             /* DEBUT DES INSTRUCTIONS */


    /* ============================ N O M ====================== */
PROCEDURE PNOM ;
 BEGIN AD:=CALCAD(RI.S1,RI.D1) ;
    DVAR:=CTX(AD) ;
    WHILE DVAR.PF=RDESC DO DVAR:=CTX(DVAR.PV) ;
    PUSHVAL(DVAR) ;
    CO:=SUCC(CO)
 END ;
          /*      */
        /* ========== ============== I D ======================== */
PROCEDURE PID ;
 BEGIN AD:=CALCAD(RI.S1,RI.D1) ;
    DVAR:=CTX(AD) ;
    IF DVAR.PF IN DIRECT
    THEN BEGIN
       IF DVAR.PF IN |<PROC,DENT,DREEL,DCH>|
       THEN DVAR.PT:=0
       ELSE
          IF DVAR.PF=FONC
          THEN DVAR.PT:=DVAR.TF ;
       DVAR.PF:=RDESC ;
       DVAR.PV:=AD
    END ;
    PUSHVAL(DVAR) ;
    CO:=SUCC(CO)
 END ;
          /*      */
    /* ============================ P O I N T E U R ============= */
PROCEDURE PPOINT ;
 BEGIN
    IF ¬ (DVAR.PF IN |<DREF,REFIND,STCOMP>|)
    THEN ERREUR(5) ;
    CASE DVAR.PF OF
    DREF :
       DTYP(1):=TYP(DVAR.PTS) ;
    REFIND,STCOMP :
       DTYP(1):=TYP(DVAR.PT)
```

```
    END ; /* CASE */
    DVAR.PF:=DTYP(1).PFEL ;
    DVAR.PT:=DTYP(1).TEL ;
    CO:=SUCC(CO)
 END ;
          /*     */
        /* ======================== I N D E X ================== */
PROCEDURE PIND /*INDEX*/ ;
 BEGIN
    IF RI.OP=INDEX
    THEN BEGIN AD:=CALCAD(RI.S2,RI.D2) ;
       DVAR:=CTX(AD) ;
       NB:=RI.N
    END
    ELSE BEGIN POPVAL(DVAR) ;
       NB:=RI.L
    END ;
    J:=1 ;
    T:=DVAR.PT ;
    IF NB > 1 THEN DESCEND(NB-1) ;
   1 : PUSHTYP(T) ; /* EMPILE LE DESC. DU TYPE TABLEAU */
       IF DTYP(TCT).PFT ¬= TAB
    THEN ERREUR(1) ;
    T:=SUCC(T) ;
    I:=1 ;
          /* PARCOURS DES INDEX */
    REPEAT
       PUSHTYP(T) ;
       T:=SUCC(T) ;
       MONTE(J) ;
       J:=J-DTYP(TCT).TE ; /* MOINS BORNE INF */
       IF J < 0 THEN ERREUR(2) ; /* J < BORNE INF */
       J:=J*DTYP(TCT).TAIL ; /* MULT PAR LA TAILLE */
       IF ((TCT=2) & (DTYP(1).TM < J))
          ! ((TCT=3) & (DTYP(2).TAIL < J))
       THEN ERREUR(2) ; /* J > BORNE SUP */
       DVAR.PV:=DVAR.PV+J ;
       I:=I+1 ;
       IF (DTYP(TCT).PFT=DVD) & (I < NB)
       THEN BEGIN
          T:=DTYP(1).TEL ; /* TABLEAU DE TABLEAU */
          TCT:=0 ;
          GOTO 1 /* BOUCLAGE */
       END
       ELSE
       IF DTYP(TCT).PFT=DVD
       THEN ERREUR(4) ;
       IF TCT ¬= 2
       THEN BEGIN
          TCT:=2 ;
          DTYP(TCT):=DTYP(TCT+1)
       END
    UNTIL I > NB ; /* CONSTRUCTION DESC. INDIRECT */
    DVAR.PF:=DTYP(1).PFEL ;
    DVAR.PT:=DTYP(1).TEL ;
```

```
    PUSHVAL(DVAR) ;
    TCT:=0 ;
    CO:=SUCC(CO)
 END ; /* IND,INDEX */
        /*     */
      /* ========================= A F F E C T E ================= */
PROCEDURE PAFF /*AFFC*/ ;
 BEGIN
    IF RI.OP=AFF THEN
    BEGIN
       AD:=CALCAD(RI.S1,RI.D1) ;
       DVAR:=CTX(AD) ;
       WHILE DVAR.PF=RDESC DO
       BEGIN
          AD:=DVAR.PV ;
          DVAR:=CTX(AD)
       END
    END ELSE
       POPVAL(DVAR) ;
      POPVAL(DP);
    CASE DVAR.PF OF
       DBOOL,DSCAL,DENTC,DREF :
       BEGIN
          AFPRIM(DVAR.DVAL) ;
          CTX(AD):=DVAR
       END ;
       BOOLIND,SCALIND,ENTCIND,REFIND :
       BEGIN
          DVAR.PF:=PRED(DVAR.PF) ;
          AFPRIM(TRAV1) ;
          CTX(DVAR.PV).VAL:=TRAV1
       END ;
       DENT,DREEL :
       BEGIN
          AFFRENT(DVAR.VALD) ;
          CTX(AD):=DVAR
       END ;
       SUBE,SUBR :
          AFSUB(DVAR.PV) ;
       STSIMP :
          AFSTRUC ;
       DPOW :
          AFPOW(DVAR.PV);
       DCH,CHIND:
            BEGIN IF ¬(DP.PF IN |<DCH,CHIND>|) THEN ERREUR(7);
            IF DP.PF =CHIND THEN DP.VALD := CTX(DP.PV).VAL;
            IF DP.VALD.NI THEN ERREUR(6);
            IF DVAR.PF = DCH
            THEN BEGIN DVAR.VALD := DP.VALD;
             CTX(AD) := DVAR
             END
            ELSE
            CTX(DVAR.PV).VALD := DP.VALD
            END
    END ; /* CASE */
```

```
    CO:=SUCC(CO)
 END ; /* AFFC,AFF */
          /*     */
       /* ======================== R E T O U R ================== */
PROCEDURE PRETOUR ;
 BEGIN CTRL:=CTX(LIBRE) ; /* CONTROLE */
    LIBRE:=PRED(LOC) ;
    DP:=CTX(LOC) ; /* MARQUE DE PROCEDURE */
    IF DP.PF=MARKF
    THEN BEGIN
       DVAR:=CTX(SUCC(LOC)) ; /* DESC. DE FONCTION */
       IF DVAR.PF IN |<SUBE,SUBP>|
       THEN DVAR.VALD:=CTX(DVAR.PV).VAL
       ELSE
          IF DVAR.PF=DPOW
          THEN PUSHVAL(CTX(DVAR.PV)) ;
       PUSHVAL(DVAR)
    END ;
    LOC:=DP.DYN ;
    SEG:=DP.SEGCT ;
    CO:=CTRL.ADRET
 END ;
          /* RETOUR      */
       /* ========================== E X E C ================ */
PROCEDURE PEXEC ;
 BEGIN PUSHCTXL(CTRL) ;
    CO:=SUCC(CO)
 END ;
       /*  EXEC  */

PROCEDURE PCHAMP (N:SHRT);       FORWARD;

     /* ====================== C H A M P DANS W I T H ============ */
PROCEDURE PCHAMW ;
 BEGIN
 DVAR:=CTX(ADSPE(RI,S2,RI.D2)) ;
 NB:=RI.N ;
 PCHAMP(3)
 END ; /* CHAMPW */
          /*     */
   /* ============================== C H A M P ============ */
PROCEDURE PCHAMP ;
 BEGIN  IF N=3 THEN GOTO 3;
    POPVAL(DVAR) ;
    NB:=RI.L ;
 3 : T:=DVAR.PT ;
    PUSHTYP(T) ; /* EMPILE LE DESC. DU TYPE RECORD */
    AD:=DVAR.PV ;
    T:=DTYP(1).TE+NB ; /* ADRESSE DU DESC DE CHAMP */
    PUSHTYP(T) ; /* EMPILE LE DESC DU TYPE DU CHAMP */
    DVAR.PV:=AD+DTYP(2).ADRL ; /* CALCUL ADRESSE */
    DVAR.PF:=DTYP(2).PFC ; /* CONSTRUIT DESC DU CHAMP */
    DVAR.PT:=DTYP(2).TP ;
    IF DTYP(2).BR <= 0
    THEN CONTROL ; /* CONTROLE SI APPARTENANCE A UNE BRANCHE */
```

```
    CO:=SUCC(CO) ;
    PUSHVAL(DVAR)
 END ; /* CHAMP */
          /*     */
     /* ============================ U N Z E R O =============== */
PROCEDURE PUNZER ;
 BEGIN
 WITH DVAR DO
 BEGIN
    PF:=RI.P ;
    DVAL := RI.V ;
    CO:=SUCC(CO)
 END
 END ; /* UNZER */
          /*     */
      /* ========================== L I T T E R A L =========== */
PROCEDURE PLITT ;
 BEGIN
 WITH DVAR DO
 BEGIN
    PF:=RI.P ;
    CASE PF OF
       DSCAL,DCH,DENTC :
          BEGIN DVAL := RI.V;
           CO:= SUCC(CO)
       END;
       DREEL,DENT :
       BEGIN
          CO:=SUCC(CO) ;
          VALO:=PROG(CO),VAL ;
          CO:=SUCC(CO)
       END ;
       LCH :
       BEGIN
         PT :=  RI.V.VEC ;
          CO:=SUCC(CO) ;
          PV:=CO ;
          CO:=CO+PT
       END
    END /* CASE */
 END;
 PUSHVAL(DVAR)
 END ; /* LITT */
          /*     */
       /*     ============================ A P P E L ============== */
PROCEDURE PAPPEL ;
 BEGIN /* PREPARE LE DESC. DE CONTROLE */
    CTRL.PF:=SEGP ; CTRL.ADRET:=SUCC(CO) ;
          /* ACCES AU DESC. DE LA PROCEDURE */
    AD:=CALCAD(RI.S2,RI.D2) ;
    DVAR:=CTX(AD) ;
    WHILE DVAR.PF=RDESC DO DVAR:=CTX(DVAR.PV) ;
          /* MISE A JOUR DU CHAINAGE STATIQUE */
    CASE RI.S2 OF
    PREDEF : DP.STA:=0 ;
```

```
    GLOBAL : DP.STA:=GLOB ;
    LOCAL : DP.STA:=LOC ;

   /* ============================== I N T 2 - I N T 3 ================

    INT2,INT3,INT4,INT5 :
       BEGIN DP:=CTX(LOC) ;
          IF DVAR.NIV ¬= PRED(DP.NV)
          THEN
          REPEAT DP:=CTX(DP.STA)
          UNTIL DVAR.NIV=PRED(DP.NV)
       END
    END ; /* CASE */
          /* CONSTRUCTION DE LA MARQUE */
IF DVAR.PF = PROC THEN DP.PF:=MARKP ELSE DP.PF:=MARKF;
DP.NV := SUCC(DVAR.NIV);
DP.SEGCT := SEG;
DP.DYN := LOC;
PUSHCTXL(DP);
IF NB ¬= DVAR.NP THEN ERREUR(10);
TRAV := SUCC(LIBRE);
LIBRE := LIBRE + DVAR.NVL;

    IF DVAR.PF=FONC
    THEN BEGIN
       DP.PF:=DVAR.PFF ;
       IF DP.PF IN|<SUBE,SUBR,DPOW>|
       THEN BEGIN
          DP.PT:=DVAR.TF ; DP.PV:=LIBRE ;
          CTX(LIBRE).VAL.NI:=TRUE ;
          LIBRE:=SUCC(LIBRE) ;
          IF DP.PF=DPOW
          THEN BEGIN
             DTYP(1):=TYP(DP.PT) ;
             FOR I:=1 TO DTYP(1).TAIL-1 DO
             BEGIN CTX(LIBRE).VAL.NI:=TRUE ;
                LIBRE:=SUCC(LIBRE)
             END
          END
       END
       ELSE
       IF DP.PF IN |<DENT,DREEL>|
       THEN DP.VALD.NI:=TRUE
       ELSE BEGIN
          DP.DVAL.NI:=TRUE ;
          DP.PTS:=DVAR.PT
       END ;
       PUSHCTXT(DP)
    END ;
          /* ADRESSE DE RETOUR */
    CO:=DVAR.ADC ;
          /*     */
          /* PASSAGE DES PARAMETRES */
```

```
    DESCEND(NB) ;
    FOR I:=1 TO NB DO
    BEGIN AD:=SUCC(AD) ;
       DVAR:=CTX(AD) ; /* DESC. DU PARAMETRE FORMEL */
   DP := CACHEVAL(EVALCT);
   EVALCT := SUCC(EVALCT);
       CASE DVAR.MODE OF
       PR,FO :
          IF DP.PF ¬= RDESC
          THEN ERREUR(16)
          ELSE PUSHCTXT(DP) ;
       REFER :
       IF DP.PF IN DIRECT
          THEN ERREUR(16)
          ELSE BEGIN DP.PT:=DVAR.TP ;
             PUSHCTXT(DP)
          END ;
       VALU :
          BEGIN WHILE DP.PF=RDESC DO DP:=CTX(DP.PV) ;
          DVAR.PF:=DVAR.PFP ;
          IF ¬ (DVAR.PF IN |<STSIMP,SUBE,SUBR,DPOW>|)
          THEN BEGIN
             IF DVAR.PF IN |<DBOOL,DSCAL,DENTC,DREF>|
             THEN BEGIN
                DVAR.PTS:=DVAR.TP ;
                AFPRIM(DVAR.DVAL)
             END
             ELSE AFFRENT(DVAR.VALD) ;
             PUSHCTXT(DVAR)
          END
          ELSE BEGIN
             DVAR.PT:=DVAR.TP ;
             LIBRE:=SUCC(LIBRE) ; DVAR.PV:=LIBRE ;
             PUSHCTXT(DVAR) ;
             CASE DVAR.PF OF
             SUBE,SUBR :
                AFSUB(LIBRE) ;
             STSIMP :
                BEGIN AFSTRUC ;
                   LIBRE:=LIBRE+DTYP(1).TAIL
                END ;
             DPOW :
                AFPOW(LIBRE)
             END /* CASE */
          END /* ELSE */
          END /* VALU */
       END /* CASE */
    END /* FOR */
 END ; /* APPEL */
          /*     */
   /* ===================== B I T ============= */
PROCEDURE PBIT ;
 BEGIN
    NB:=RI.L ; /* NOMBRE D'ELEMENTS DE L'ENSEMBLE A CONSTRUIRE */
    DP.VAL.NI:=FALSE ;
```

```
    DP.VAL.VP:= |< >| ; /* INITIALISE A VIDE */
    FOR I:=1 TO NB DO
    BEGIN
       POPVAL(DVAR) ; /* DEPILE LES EXPRESSIONS */
       CONVENT ;
       IF J > VALMAX THEN ERREUR(5) ;
       DP.VAL.VP:=DP.VAL.VP | |<J>|
    END ; /* FOR */
    DVAR.PF:=DPROW ;
    DVAR.PV:=0 ;
    PUSHVAL(DP) ; /* EMPILE SUCCESSIVEMENT LA VALEUR */
    PUSHVAL(DVAR) ; /* ET LE DESCRIPTEUR */
    CO:=SUCC(CO)
 END ; /* BIT */
          /*     */
    /* ================ P L U S , M O I N ... ======= */
PROCEDURE PPLUS /*MOIN,MULT,DIVENT,MODULO*/ ;
 BEGIN
    POPVAL(DP) ; /* DEUXIEME OPERANDE */
    POPVAL(DVAR) ; /* PREMIER OPERANDE */
    IF DVAR.PF IN  |<DENT,SUBE,DREEL,SUBR>| THEN
    BEGIN
       IF DVAR.PF IN  |<SUBE,SUBR>| THEN
       BEGIN
          DVAR.VALD:=CTX(DVAR.PV).VAL ;
          DVAR.PF:=PRED(DVAR.PF) /* SUBE/SUBR => DENT/DREEL */
       END ;
       IF DVAR.VALD.NI THEN ERREUR(6) ;
       IF (DVAR.PF=DENT) & (DP.PF IN  |<DREEL,SUBR>|)
       THEN BEGIN
          DVAR.VALD.VR:=DVAR.VALD.VE ; /* ENTIER => REEL */
          DVAR.PF:=DREEL
       END ;
       AFFRENT(DP.VALD)
    END ELSE
       BEGIN
       IF DVAR.PF=ENTCIND THEN
       DVAR.DVAL:=CTX(DVAR.PV).VAL ;
       IF DVAR.DVAL.NI THEN ERREUR(6) ;
       IF DP.PF IN  |<DREEL,SUBR>| THEN
       BEGIN
          DVAR.VALD.VR:=DVAR.DVAL.VEC ; /* ENTC => REEL */
          DVAR.PF:=DREEL
       END ELSE BEGIN
          DVAR.PF:=DENT ;
          DVAR.VALD.VE:=DVAR.DVAL.VEC /* ENTC => ENTIER */
       END ;
       AFFRENT(TRAV1) ;
       DP.VALD:=TRAV1
    END ;
    CASE RI.OP OF
       PLUS : IF DVAR.PF=DENT THEN
       DVAR.VALD.VE:=DVAR.VALD.VE+DP.VALD.VE
          ELSE DVAR.VALD.VR:=DVAR.VALD.VR+DP.VALD.VR ;
       MOIN : IF DVAR.PF=DENT THEN
```

```
            DVAR.VALD.VE:=DVAR.VALD.VE-DP.VALD.VE
            ELSE DVAR.VALD.VR:=DVAR.VALD.VR-DP.VALD.VE ;
         MULT : IF DVAR.PF=DENT THEN
            DVAR.VALD.VE:=DVAR.VALD.VE*DP.VALD.VE
            ELSE DVAR.VALD.VR:=DVAR.VALD.VR*DP.VALD.VR ;
         DIVENT : DVAR.VALD.VE:=DVAR.VALD.VE DIV DP.VALD.VE ;
         MODULO : DVAR.VALD.VE:=DVAR.VALD.VE MOD DP.VALD.VE
      END ; /* CASE */
      PUSHVAL(DVAR) ;
      CO:=SUCC(CO)
   END ; /* PLUS,MOIN,MULT,DIVEN,MOD */
            /*      */
     /* ================== T R U N C ================ */
PROCEDURE PTRONK ;
   BEGIN        /*
      POPVAL(DVAR) ;
      DP.PF:=DENT ;
      DP.VALD.NI:=FALSE ;
      IF DVAR.PF=SUBR THEN
      DVAR.VALD:=CTX(DVAR.PV).VAL ;
      IF DVAR.VALD.NI THEN ERREUR(6) ;
      DP.VALD.VE:=TRUNC(DVAR.VALD.VR) ;
      PUSHVAL(DP) ;
      CO:=SUCC(CO)          */
   END ; /* TRUNC */
            /*      */
       /* ================== D I V I S I O N ============ */
PROCEDURE PDIVIZ ;
   BEGIN
   POPVAL(DP) ; /* 2-EME OPERANDE */
   DVAR.PF:=REEL ;
   AFFRENT(DVAR.VALD) ; /* DANS DVAR */
   POPVAL(DP) ;
   AFFRENT(DP.VALD) ; /* 1-ER OPERANDE DANS DP */
   DVAR.VALD.VR:=DP.VALD.VR/DVAR.VALD.VR
   END ; /* DIV */
            /*      */
      /* ================== A B S , N E G A ========== */
PROCEDURE PABSOLU  /*NEGA*/ ;
   BEGIN
      POPVAL(DVAR) ;
      CONVENTRE;
      CASE DVAR.PF OF
         DENT : IF RI.OP=NEGA THEN
            DVAR.VALD.VE:=-DVAR.VALD.VE
            ELSE
            IF DVAR.VALD.VE <0 THEN DVAR.VALD.VE := - DVAR.VALD.VE ;
         REEL : IF RI.OP=NEGA THEN
            DVAR.VALD.VR:=-DVAR.VALD.VR
            ELSE
            IF DVAR.VALD.VR <0 THEN DVAR.VALD.VR := -DVAR.VALD.VR
      END ; /* CASE */
      PUSHVAL(DVAR) ;
      CO:=SUCC(CO)
   END ; /* ABS,NEG */
```

```
          /*      */
       /* ================ S U C C , P R E D ========*/
PROCEDURE PSUC /*PRE*/ ;
 BEGIN
    POPVAL(DVAR) ;
    CONVENT ; /* RESULTAT DANS J */
    IF RT,OP=SUC THEN
    J:=J+1
    ELSE J:=J-1 ;
    IF DVAR.PF IN INDIR THEN
    DVAR.PF:=PRED(DVAR.PF) ;
    CASE DVAR.PF OF
       DENT : DVAR.VALD.VE:=J ;
       DSCAL : DVAR.DVAL.VS:= J  ;
       DENTC : DVAR.DVAL.VEC:=J ;
       DBOOL : IF J > 1 THEN
       ERREUR(8)
       ELSE IF J=0 THEN
       DVAR.DVAL.VB:=FALSE
       ELSE DVAR.DVAL.VB:=TRUE
    END ;
    PUSHVAL(DVAR) ;
    CO:=SUCC(CO)
 END ; /* SUC,PRE */
          /*      */
     /* ================= O U , E T ================ */
PROCEDURE POU /*ET*/ ;
 BEGIN
    POPVAL(DP) ; /* 2-EME OPERANDE */
    POPVAL(DVAR) ; /* 1-ER OPERANDE */
    IF DVAR.PF IN  |<DBOOL,BOOLIND>| THEN
    BEGIN
       IF DVAR.PF=BOOLIND THEN
       BEGIN
          DVAR.PF:=DBOOL ;
          DVAR.DVAL:=CTX(DVAR.PV).VAL
       END ;
       IF DP.PF=BOOLIND THEN
       DP.DVAL:=CTX(DP.PV).VAL ;
       IF DVAR.DVAL.NI | DP.DVAL.NI THEN
       ERREUR(6) ;
       IF RT.OP=OU THEN
       DVAR.DVAL.VB:=DVAR.DVAL.VB | DP.DVAL.VB
       ELSE DVAR.DVAL.VB:=DVAR.DVAL.VB & DP.DVAL.VB ;
       PUSHVAL(DVAR)
   END
    ELSE BEGIN
       IF DP.PF <> DPOW THEN ERREUR(7) ;
       ACCEPOW ;
       IF RT.OP=OU THEN DVAR.VAL.VP:=DVAR.VAL.VP | TRAV1.VP
       ELSE DVAR.VAL.VP:=DVAR.VAL.VP & TRAV1.VP ;
       PUSHVAL(DVAR) ;
       DP.PV:=0 ;
       PUSHVAL(DP)
    END ;
```

```
    CO:=SUCC(CO)
 END ; /* OU,ET */
          /*     */
     /* ================== N O N ================ */
PROCEDURE PNON ;
 BEGIN
    POPVAL(DVAR) ;
    IF ¬ (DVAR.PF IN  |<BOOLIND,DBOOL>|)
    THEN ERREUR(7) ;
    IF DVAR.PF=BOOLIND THEN
    BEGIN
       DVAR.PF:=DBOOL ;
       DVAR.DVAL:=CTX(DVAR.PV).VAL
    END ;
    IF DVAR.DVAL.NI THEN ERREUR(6) ;
    DVAR.DVAL.VB:= ¬ DVAR.DVAL.VB ;
    PUSHVAL(DVAR) ;
    CO:=SUCC(CO)
 END ; /* NON */
          /*     */
   /* ============== C O M P A R A I S O N S ======= */
PROCEDURE PEG /*DIFFER,SUP,SUPEG,INF,INFEG*/ ;
 BEGIN
 POPVAL(DP) ; POPVAL(DVAR) ;
 IF (DP.PF ¬= DPOW) & ( ¬ (DP.PF IN  |<DVAR.PF,SUCC(DVAR.PF)>|))
 THEN ERREUR(7) ;
 TRAV1.VB:=FALSE ;
 CASE DVAR.PF OF
    DENT : BEGIN
       14 : IF DP.PF=SUBE THEN
          DP.VALD:=CTX(DP.PV).VAL ;
          IF DP.VALD.NI | DVAR.VALD.NI THEN ERREUR(6) ;
          K:=DVAR.VALD.VE >= DP.VALD.VE ;
          TRAV1.VB:=DVAR.VALD.VE ¬= DP.VALD.VE
       END ;
    DCH : BEGIN
       145 : IF DP.PF=CHIND THEN DP.VALD:=CTX(DP.PV).VAL;
          IF DP.VALD.NI | DVAR.VALD.NI THEN ERREUR(6);
          K := DVAR.VALD.VCH >= DP.VALD.VCH;
          TRAV1.VB := DVAR.VALD.VCH ¬= DP.VALD.VCH
       END;
    DREEL : BEGIN
       15 : IF DP.PF=SUBR THEN
          DP.VALD:=CTX(DP.PV).VAL ;
          IF DP.VALD.NI | DVAR.VALD.NI THEN ERREUR(6) ;
          K:=DVAR.VALD.VE >= DP.VALD.VE ;
          TRAV1.VB:=DVAR.VALD.VR ¬= DP.DVAL.VR
       END ;
    DBOOL : BEGIN
       16 : IF DP.PF=BOOLIND THEN
          DP.DVAL:=CTX(DP.PV).VAL ;
          IF DP.DVAL.NI | DVAR.DVAL.NI THEN ERREUR(6) ;
          K:=DVAR.DVAL.VB >= DP.DVAL.VB ;
          TRAV1.VB:=DVAR.DVAL.VB ¬= DP.DVAL.VB
       END ;
```

```
DENTC : BEGIN
   17 : IF DP.PF=ENTCIND THEN
        DP.DVAL:=CTX(DP.PV).VAL ;
        IF DP.DVAL.NI | DVAR.DVAL.NI THEN ERREUR(6)    ;
        K:=DVAR.DVAL.VE >= DP.DVAL.VEC ;
        TRAV1.VB:=DVAR.DVAL.VEC ¬= DP.DVAL.VEC
   END ;
DSCAL : BEGIN
   18 : IF DP.PF=SCALIND THEN DP.DVAL:=CTX(DP.PV).VAL ;
        IF DP.DVAL.NI | DVAR.DVAL.NI THEN ERREUR(6) ;
        K:=DVAR.DVAL.VS >= DP.DVAL.VS ;
        TRAV1.VB:=DVAR.DVAL.VS ¬= DP.DVAL.VS
   END ;
DREF : BEGIN
   19 : IF DP.PF=REFIND THEN
        DP.DVAL:=CTX(DP.PV).VAL ;
        IF DP.DVAL.NI | DVAR.DVAL.NI THEN ERREUR(6) ;
        K:=DVAR.DVAL.VS >= DP.DVAL.VS ;
        TRAV1.VB := DVAR.DVAL.VAD ¬= DP.DVAL.VAD
   END ;
SUBS : BEGIN
        DVAR.VALD:=CTX(DVAR.PV).VAL ;
        GOTO 14
   END ;
CHIND : BEGIN
          DVAR.VALD := CTX(DVAR.PV).VAL;
          GOTO 145
   END;
SUBR : BEGIN
        DVAR.VALD:=CTX(DVAR.PV).VAL ;
        GOTO 15
   END ;
BOOLIND : BEGIN
        DVAR.DVAL:=CTX(DVAR.PV).VAL ;
        GOTO 16
   END ;
ENTCIND : BEGIN
        DVAR.DVAL:=CTX(DVAR.PV).VAL ;
        GOTO 17
   END ;
SCALIND : BEGIN
        DVAR.DVAL:=CTX(DVAR.PV).VAL ;
        GOTO 18
   END ;
REFIND : BEGIN
        DVAR.DVAL:=CTX(DVAR.PV).VAL ;
        GOTO 19
   END ;
STSIMP : BEGIN
        I:=0;
        DTYP(1):=TYP(DVAR.PT) ;
        IF (DP.PF ¬= LCH) & (DP.PT ¬= DVAR.PT) THEN ERREUR(7) ;
        J:=DTYP(1).TAIL ;
        IF DP.PF=LCH THEN
        IF J > DP.PT THEN J:=DP.PT ;
```

```
        REPEAT
        IF DP.PF ¬= LCH THEN
        BEGIN
        TRAV1.VB:= Z(CTX(DVAR.PV+I).VAL , CTX(DP.PV+I).VAL , EG);
        K:= Z(CTX(DVAR.PV+I).VAL , CTX(DP.PV+I).VAL , SUPEG)
        END ELSE
        BEGIN
        TRAV1.VB:= Z(CTX(DVAR.PV+I).VAL , PROG(DP.PV+I).VAL , DIFFER);
        K:= Z(CTX(DVAR.PV+I).VAL , PROG(DP.PV+I).VAL , SUPEG)
        END ;
        I:=I+1
        UNTIL (I=J) | TRAV1.VB ;
        IF (¬ TRAV1.VB) & (DP.PF=LCH) & (DTYP(1).TAIL ¬= J) THEN
       TRAV1.VB := TRUE
    END ;
LCH : BEGIN
        I:=0 ; J:=DVAR.PT ;
        IF DP.PF ¬= LCH THEN
        BEGIN
        DTYP(1):=TYP(DP.PT) ;
        IF DTYP(1).TAIL < J THEN J:=DTYP(1).TAIL ;
        IF DTYP(1).TEL ¬= 0 THEN ERREUR(7)
        END ELSE
        IF DP.PT < J THEN J:=DP.PT ;
        REPEAT
        IF DP.PF ¬= LCH THEN
        BEGIN
        TRAV1.VB:= Z(PROG(DVAR.PV+I).VAL , CTX(DP.PV+I).VAL , DIFFER);
        K:= Z(PROG(DVAR.PV+I).VAL , CTX(DP.PV+I).VAL , SUPEG)
        END ELSE
        BEGIN
        TRAV1.VB:= Z(PROG(DVAR.PV+I).VAL , PROG(DP.PV+I).VAL , DIFFER);
        K:= Z(PROG(DVAR.PV+I).VAL , PROG(DP.PV+I).VAL , SUPEG)
        END ;
        I:=I+1
        UNTIL (I=J) | TRAV1.VB ;
        IF ( ¬ TRAV1.VB) & (J < DVAR.PT) THEN
        TRAV1.VB:=TRUE
        ELSE K:=FALSE
    END ;
DPOW : BEGIN
        ACCEPOW ;
        DP.VAL:=TRAV1 ;
        TRAV1.VB:=DVAR.VAL.VP=DP.VAL.VP ;
        K:=DVAR.VAL.VP >= DP.VAL.VP
    END
END ; /* CASE */
CASE RI.OP OF
    EG : K:=TRAV1.VB ;
    DIFFER : K:=TRAV1.VB ;
    SUP : K:=TRAV1.VB & K ;
    SUPEG : /* K:=K */ ;
    INF : K:=K & TRAV1.VB ;
    INFEG : K:=K | TRAV1.VB
END ;
```

```
    DVAR.PF:=DBOOL ;
    DVAR.DVAL.NI:=FALSE ;
    DVAR.DVAL.VB:=K ;
    PUSHVAL(DVAR) ;
    CO:=SUCC(CO)
 END ; /* COMPARAISON */
           /*      */
     /* ====================== I N ================== */
PROCEDURE PINN ;
 BEGIN
    POPVAL(DP) ;
    IF DP.PF ¬= DPOW THEN ERREUR(7) ;
    IF DP.PV ¬= 0 THEN DP:=CTX(DP.PV)
    ELSE POPVAL(DP) ;
    POPVAL(DVAR) ;
    CONVENT ;
    IF J > VALMAX THEN ERREUR(5) ;
    DVAR.PF:=DBOOL ;
    DVAR.DVAL.NI:=FALSE ;
    DVAR.DVAL.VB:=J IN DP,VAL,VP ;
    PUSHVAL(DVAR) ;
 CO:=SUCC(CO)
 END ; /* IN */
           /*      */
    /* ==================== E X I T ================ */
PROCEDURE PEXIT ;
 BEGIN AD:=CALCAD(RI,S1,RI.D1) ;
    DVAR:=CTX(AD) ;
    IF RI,S1=GLOBAL
    THEN I:=1
    ELSE I:=INT(PRED(RI.S1)) ;
    REPEAT LOC:=CTX(LOC).DYN
    UNTIL INT(CTX(LOC).NV)=I ;
    DP:=CTX(LIBRE) ;
    WHILE(DP.ADS1 ¬= DVAR.PV) & (DP.PF=SEGP) DO
       PULLCTXL(DP) ;
    IF(DP.PF=SEGP) & (DVAR.PT=0)
    THEN ERREUR(3) ;
    CO:=DVAR.PV+DVAR.PT
 END ;
           /*      */
PROCEDURE PCREP ;
 BEGIN /* CREATION D'UN DESC. DE PROCEDURE */
    NB:=RI.L ;
    FOR I:=1 TO NB DO
    BEGIN CO:=SUCC(CO) ;
       PUSHCTXT(PROG(CO).DESCRIM)
    END ;
    CO:=SUCC(CO)
 END ;
           /*      */
PROCEDURE PCREST ;
 BEGIN /* CREATION D'UN DESC. POUR TYPES STANDARDS */
    NB:= RI.V.VEC ;
    DVAR.PF:=RI.P ;
```

```
     IF RI.P=DBOOL
     THEN DVAR.DVAL.NI:=TRUE
     ELSE DVAR.VALD.NI:=TRUE ;
     FOR I:=1 TO NB DO
        PUSHCTXT(DVAR) ;
     CO:=SUCC(CO)
 END ;
            /*      */
PROCEDURE PCREE ;
 BEGIN NB:=RI.NB ;
     DVAR.PF:=RI.PX ;
     IF RI.PX IN |<DSCAL,DENTC,DREF>|
     THEN BEGIN
        DVAR.PTS:=RI.TP ;
        DVAR.DVAL.NI:=TRUE ;
        FOR I:=1 TO NB DO
           PUSHCTXT(DVAR)
     END
     ELSE BEGIN
        DVAR.PT:=RI.TP ;
        DVAR.PV:=SUCC(LIBRE) ;
        PUSHCTXT(DVAR) ;
        DP.VAL.NI:=TRUE ;
        CASE DVAR.PF OF
        DPOW,SUBE,SUBR :
           FOR I:=1 TO NB DO PUSHCTXL(DP) ;
        STSIMP :
           BEGIN DTYP(1):=TYP(DVAR.PT) ; L:=DTYP(1).TAIL ;
           FOR I:=1 TO NB DO
           FOR J:=1 TO L DO
           PUSHCTXL(DP) ;
           END
        END /* CASE */
     END ; /* ELSE */
     CO:=SUCC(CO)
 END ;
            /*      */
      /* ==================== R E S E T =============== */
PROCEDURE PRES ;
 BEGIN
     DVAR:=CTX(CALCAD(RI.S1,RI.D1)) ; /* S,D DE LA VARIABLE CLASSE */
     POPVAL(DP) ; /* DESC DE LA VARIABLE POINTEUR */
     IF DP.PF=REFIND THEN
     DP.DVAL:=CTX(DP.PV).DVAL ;
     IF DP.DVAL.NI THEN ERREUR(6) ;
     CTX(DVAR.PV).VAL:=DP.DVAL
 END ; /* RES */
            /*      */
            /*      */
        /* INSTRUCTIONS DE CONTROLE */
            /*      */
            /*      */
      /* ===================== I F ================ */
PROCEDURE PIFS ;
 BEGIN POPVAL(DVAR) ;
```

```
    TEST ;
    IF DVAR.DVAL.VB
    THEN BEGIN
       CTRL.PF:=ALT ;
       CTRL.ADS1:=CO+RI.L ;
       PUSHCTXL(CTRL) ;
       CO:=SUCC(CO)
    END
    ELSE
       CO:=CO+RI.L
 END ;
          /*      */
      /* ================ I F - E L S E ============== */
PROCEDURE PIFELSE ;
 BEGIN POPVAL(DVAR) ;
    TEST ;
    CTRL.ADS1:=CO+RI.M1 ;
    CTRL.PF:=ALT ;
    PUSHCTXL(CTRL) ;
    IF DVAR.DVAL.VB
    THEN CO:=SUCC(CO)
    ELSE CO:=CO+RI.M2
 END ;
          /*      */
    /* ==================== B O U C L E ============= */
PROCEDURE PBOUC ;
 BEGIN CO:=SUCC(CO) ;
    CTRL.ADS1:=CO ;
    CTRL.PF:=BOUK ;
    PUSHCTXL(CTRL)
 END ;
          /*      */
     /* ===================== C Y C L E ============= */
PROCEDURE PCYCLE ;
 BEGIN CTRL:=CTX(LIBRE) ;
    CO:=CTRL.ADS1
 END ;
          /*      */


   /* ================================== I N T 3 - I N T 4 ============== */


    /* ==================== U N T I L ============= */
PROCEDURE PUNTI ;
 BEGIN POPVAL(DVAR) ;
    TEST ;
    IF DVAR.DVAL.VB
    THEN BEGIN
       LIBRE:=PRED(LIBRE) ;
       CO:=SUCC(CO)
    END
    ELSE BEGIN
       CTRL:=CTX(LIBRE) ;
       CO:=CTRL.ADS1
```

```
    END
 END ;
          /*      */
    /* ==================== C A S E ================== */
PROCEDURE PKASE ;
 BEGIN CTRL.ADS1:=CO+RI.L ;
    CTRL.PF:=ALT ;
    PUSHCTXL(CTRL) ;
    POPVAL(DVAR) ;
    CONVENT ; /* VALEUR DE L'EXPRESSION */
    CO:=SUCC(CO)
 END ;
          /*      */
    /* ==================== G O T O ================== */
PROCEDURE PGO ;
 BEGIN NB:=INT(RI.S1) ;
       /* DEPILE NB DESCRIPTEURS DE SEGMENT */
    LIBRE:=LIBRE-NB ;
    CO:=CO+RI.D1
 END ;
          /*      */
     /* ==================== F I N ================== */
PROCEDURE PFIN ;
 BEGIN CTRL := CTX(LIBRE);
     LIBRE := PRED(LIBRE);
    CO:=CTRL.ADS1
 END ;
          /*      */
      /* ==================== S O R T I E ============= */
PROCEDURE PSORT ;
 BEGIN CTRL:=CTX(LIBRE) ;
    IF CTRL.PF=ALT
    THEN LIBRE:=PRED(LIBRE)
    ELSE /* SORTIE D'UN SEGMENT WITH */
    BEGIN CTRL:=CTX(SEG) ;
       LIBRE:=PRED(SEG) ;
       SEG:=CTRL.AFS2
    END ;
    CO:=SUCC(CO)
 END ;
          /*      */
    /* ==================== B D O ================== */
PROCEDURE PBDO ;
 BEGIN POPVAL(DVAR) ;
    TEST ;
    IF DVAR.DVAL.VB
    THEN CO:=SUCC(CO)
    ELSE BEGIN
       LIBRE:=PRED(LIBRE) ;
       CO:=CO+RI.L
    END
 END ;
          /*      */
     /*  ==================== C F S ================== */
PROCEDURE PCFS ;
```

```
BEGIN NB:=RI.M1 ;
    FOR I:=1 TO NB DO
        IF J=PROG(CO+I).VAL.VS
        THEN GOTO 5 ;
    /* NE TROUVE PAS */
    IF RI.M2=0
    THEN ERREUR(15) /* Y A PLUS DE CAS */
    ELSE CO:=CO+RI.M2 ;
    GOTO 6 ;
5 : CO:=CO+NB+1 ; /* TROUVE */
6 :
END ;
          /*      */
     /* ==================== W I T H ================= */
PROCEDURE PWT ;
BEGIN
    NB := PI.L ;
    CTRL.PF:=WTH ;
    CTRL.ADS2:=CO ;
    CTRL.AFS2:=SEG ;
    PUSHCTXL(CTRL) ;
    SEG:=LIBRE ; /* SEG POINTE SUR LE DESC DE WITH */
    FOR I:=1 TO NB DO
    BEGIN
        POPVAL(DVAR) ;
        PUSHCTXL(DVAR) /* EMPILE LES DESC DES VARIABLES MISES EN
        FACTEUR */
    END
END ; /* WT */
          /*      */
     /* ==================== F O R ============== */
PROCEDURE PFORUP /*FORBOT*/ ;
BEGIN
    POPVAL(DVAR) ;
    CONVENT ;
    DP.PF:=DENT ;
    DP.DVAL.NI:=FALSE ;
    DP.DVAL.VE:=J ;
    POPVAL(DVAR) ;
    CONVENT ;
    K:=(RI.OP=FORUP) ;
    IF (K & (J > DP.VALD.VE)) | ((¬ K) & (DP.VALD.VE > J))
    THEN CO:=CO+RI.N
    ELSE BEGIN
        PUSHCTXL(DP) ;
        LIBRE:=SUCC(LIBRE) ;
        CO:=SUCC(CO) ;
        CTRL.ADS1:=CO ;
        CTRL.PF:=RI.OP ;
        CTX(LIBRE):=CTRL ;
        AD:=CALCAD(RI.S2,RI.D2) ;
        DP.PF:=DENT ;
        DP.VALD.NI:=FALSE ;
        DP.VALD.VE:=J ;
        DVAR:=CTX(AD) ;
```

```
        AFBOUC
     END
 END ; /* FOR */
           /*      */
     /* ====================== T O ====================== */
PROCEDURE PTOU ;
 BEGIN
     CTRL:=CTX(LIBRE) ;
     DP:=CTX(PRED(LIBRE)) ; /* BORNE SUP (OU INF) DE LA BOUCLE */
     DVAR:=CTX(CALCAD(RI.S1,RI.D1)) ; /* DESC DE LA VARIABLE DE
     CONTROLE */
     CONVENT ;
     K:=(CTRL.PF=FORTO) ;
     IF K THEN J:=J+1 ELSE J:=J-1 ;
     IF (K & (J > DP.VALD.VE)) | (( ¬ K) & (DP.VALD.VE > J)) THEN
     BEGIN
        CO:=SUCC(CO) ;
        LIBRE:=PRED(LIBRE)
     END ELSE
     BEGIN
        DP.VALD.VE:=J ;
        AFBOUC ;
        CO:=CTRL.ADS1
     END
 END ; /* TOU */
           /*      */
      /* ======================== S T O P ================ */
PROCEDURE PSTOP ;
 BEGIN
     ARRET:=FALSE
 END;
           /*      */
      /* ================== A L L O C ================ */
PROCEDURE PALOC ;
 BEGIN
     NB:=RI.N ;
     DP:=CTX(CALCAD(RI.S2,RI.D2)) ; /* S2,D2 : DESC DE LA CLASSE */
    POPVAL(DVAR) ; /* DESC DE LA VARIABLE POINTEUR */
     WHILE DVAR.PF=RDESC DO
     BEGIN
        AD:=DVAR.PV ;
        DVAR:=CTX(AD)
     END ;
     T:=DVAR.PT ;
     TRAV1.VAD:=SUCC(DP.PV) ; /* ADRESSE DEBUT CLASS */
     PUSHTYP(DP.PT) ; /* TYPE CLASSE */
     PUSHTYP(T) ;
     L:=DP.PV ;
     J:=DTYP(2).TAIL ; /* TAILLE MAX DU RECORD */
     DP.VAL:=CTX(L).VAL ; /* ADRESSE LIBRE DANS CLASS */
     IF DP.VAL.VAD+J > TRAV1.VAD+DTYP(1).TAIL THEN /* CLASSE PLEINE */
     TRAV1.VAD:=0 ELSE TRAV1.VAD:=DP.VAL.VAD ;
     IF DVAR.PF=REFIND THEN
     BEGIN
        AD:=DVAR.PV ;
```

```
      DVAR.VAL.NI:=FALSE ;
      DVAR.VAL.VAD:=TRAV1.VAD
   END ELSE
   BEGIN
   DVAR.DVAL.NI:=FALSE ;
      DVAR.DVAL.VAD:=TRAV1.VAD
   END ;
   CTX(AD):=DVAR ;
   IF TRAV1.VAD=0 THEN GOTO 50 ;
   IF NB ¬= 0 THEN
   BEGIN
      DESCEND(NB) ;
      IF DTYP(2).TE=0 THEN ERREUR(12) ;
      PUSHTYP(DTYP(2).TE) ;
      I:=1 ;
      FOR I:=1 TO NB DO
      BEGIN /* BOUCLE SUR LES TAGFIELDS IMBRIQUES */
      POPVAL(DVAR) ;
         TESTAG ;
         IF K THEN IF ((DTYP(3).FILS=0) & (I ¬= NB)) THEN
         ERREUR(12)
         ELSE DTYP(3):=TYP(DTYP(3).FILS)
         ELSE
         BEGIN
         J:=J-DTYP(3).CORT ;
         IF DTYP(3).FRER=0 THEN ERREUR(13) ;
         DTYP(3):=TYP(DTYP(3).FRER)
         END
      END /* FOR */
   END ; /* NB ¬= 0 */
   DP.VAL.VAD:=DP.VAL.VAD+J ;
   CTX(L):=DP   ;
50 : END;  /* ALOC */
   -      /*      */
          /*================= F S P F ==================*/
PROCEDURE PFSPE;
 BEGIN
 CASE RI.L OF
  PGET: BEGIN
     GET(INPUT);
     POPVAL(DVAR);
     DP.PF:=VALEUR;
     DP.VAL.NI:=FALSE;
     DP.VAL.PFV:=CH;
     FOR I:=0 TO 79 DO
      BEGIN DP.VAL.VCH:=INPUT"(I);
       CTX(DVAR.PV+I):=DP
      END
     END;
  PPUT: BEGIN
     POPVAL(DVAR);
     FOR I:=0 TO 132 DO
     BEGIN
     DP:=CTX(DVAR.PV+I);
     OUTPUT"(I):=DP.VAL.VCH
```

```
    END;
    PUT(OUTPUT)
   END;
  FINT: BEGIN
    POPVAL(DVAR);
    IF ¬(DVAR.PF IN [<DCH,CHIND>]) THEN
    ERREUR(7);
    IF DVAR.PF=CHIND THEN DVAR.VALD:=CTX(DVAR.PV).VAL;
    IF DVAR.VALD.NI THEN ERREUR(6);
    DVAR.PF:=DENT;
    DVAR.VALD.PFV:=ENT;
    DVAR.VALD.VE:=INT(DVAR.VALD.VCH);
    PUSHVAL(DVAR)
    END;
  FCHR: BEGIN
    POPVAL(DVAR);
    IF ¬(DVAR.PF IN [<DENT,SUBE,DENTC,ENTCIND>]) THEN ERREUR(7);
    CONVENTRE;
    IF DVAR.VALD.NI THEN ERREUR(6);
    DVAR.PF:=DCH;
    DVAR.VALD.PFV:=CH;
    DVAR.VALD.VCH:=CHR(DVAR.VALD.VE);
    PUSHVAL(DVAR)
   END
  END; /* CASE */
 CO := SUCC(CO)
END;  /* PFSPE */
          /*     */


      /* ========================== M A I N ====================== */


BEGIN       /* PROGRAMME PRINCIPAL */


    DIRECT:=  [<DENT,DREEL,DBOOL,DSCAL,DENTC,DREF,DCH,PROC,FONC>] ;
    INDIR:=  [<SUBE,SUBR,BOOLIND,SCALIND,ENTCIND,REFIND,CHIND,RDESC>] ;
    CHARGE ;
    RI:=PROG(0) ;
    CO:=RI.L ; /* POINT D'ENTREE DU PROGRAMME */
    ARRET:=FALSE ;
          /*     */
          /* DEBUT DE L'INTERPRETATION */

 REPEAT
  RI:=PROG(CO) ; /* FETCH */
  CASE RI.OP OF /* DECODE */

   NOM : PNOM ;
   ID : PID ;
   POINT : PPOINT ;
   IND,INDEX : PIND ;
   AFF,AFFC : PAFF ;
   RETOUR : PRETOUR ;
```

```
   EXEC : PEXEC ;
   CHAMW : PCHAMW ;
   CHAMP : PCHAMP(1) ;
   UNZER : PUNZER ;
   LITT : PLITT ;
   APPEL : PAPPEL ;
   BIT : PBIT ;
   PLUS,MOIN,MULT,DIVENT,MODULO : PPLUS ;
   TRONK : PTRONK ;
   DIVIZ : PDIVIZ ;
   ABSOLU,NEGA : PABSOLU ;
   SUC,PRE : PSUC ;
   OU,ET : POU ;
   NON : PNON ;
   EG,DIFFER,SUP,SUPEG,INFEG : PEG ;
   INN : PINN ;
   CREP : PCREP ;
   CREST : PCREST ;
   CREE : PCREE ;
   PES : PPES ;
   TES : PTES ;
   TESELSE : PTESELSE ;
   BOUC : PBOUC ;
   CYC : PCYCLE ;
   UNTT : PUNTI ;
   KASE : PKASE ;
   GO : PGO ;
   EXIT : PEXIT ;
   FIN : PFIN ;
   SORT : PSORT ;
   BOO : PBOO ;
   OES : POES ;
   WT : PWT ;
   FORUP,FORBOT : PFORUP ;
   TOU : PTOU ;
   STOP : PSTOP ;
   ESPE : PESPE ;
   ALOC : PALOC
  END  /* CASE DECODE */

 UNTIL ARRET ;

 100 : FICHIER":='FIN INTERPRETATION'

 END.
/*
//
```

ANNEXE VI

TOY - SYNTAXE ET FONCTIONS SEMANTIQUES

```
<PROGRAMME>     *GENERER* 'DECLARE' <DECLISTE> *GENERER* 'PVG'
                                    <INSTLISTE> *GENERER* 'FIN'

<DECLISTE>      <DECLARATION>
                <DECLARATION> 'VG' <DECLISTE>

<DECLARATION>   *GENERER* 'NOMBRE' *RANGER* 'IDENT' <INITIALISE>
                *GENERER* 'ENTREE' *RANGER* 'IDENT' 'EGAL' *GENERER*
                                                    'ENTIER'
                *GENERER* 'SORTIE' *RANGER* 'IDENT' 'EGAL' *GENERER*
                                                    'ENTIER'
                *GENERER* 'ETIQ' *RANGER* 'IDENT'

<INITIALISE>    *PUSH* 'EGAL' *GENERER* 'ENTIER' *PULL*
                ()

<INSTLISTE>     <INSTRUCTION>
                <INSTRUCTION> *GENERER* 'PVG' <INSTLISTE>

<INSTRUCTION>   *ASSOCIER* 'IDENT' <EXPOUETIQ>
                *GENERER* 'ENTIER' <EXPRESSION>

<EXPOUETIQ>     *GENERER* 'DPT' <INSTRUCTION>
                <EXPRESSION>
                ()

<EXPRESSION>    <COMPARAISON> <SUITEXP>
                <OPERARITH> <SUITEXP>
                <AFFECTATION> <SUITEXP>

<SUITEXP>       <EXPRESSION>
                *PUSH* 'INTERRO' *ASSOCIER* 'IDENT' *PULL*
                ()

<COMPARAISON>   *PUSH* <OPDECOMP> <PRIMAIRE> *PULL*

<OPERARITH>     *PUSH* <OPDEARITH> <PRIMAIRE> *PULL*

<AFFECTATION>   *PUSH* 'AFFECT' *ASSOCIER* 'IDENT' *PULL*

<PRIMAIRE>      *ASSOCIER* 'IDENT'
                *GENERER* 'ENTIER'

<OPDECOMP>      'EGAL'
                'SUPER'
                'INFEGAL'

<OPDEARITH>     'PLUS'
                'MOINS'
```

INSTRUCTIONS ET INTERPRETATION

Format de l'information:

```
        Type INFO = record

case Prefixe:(Desc, Val, inst, indef) of.
            desc: (record TYP:(nbre, Ent, Sort, étiq) ;
                          AD : Adresse);
            val : (V:integer) ;
            inst: (OP:(declare, nombre, ....., fin)) ;
            modif:
end
```

"Declare"	initialise l'interprétation
'Nombre' 0	crée un descripteur de nombre non initialisé
'Entrée'n	crée un descripteur d'entrée voir logique n
'Sortie' n	crée un descripteur de sortie
'Etiqu' e	crée un descripteur d'étiquette initialisé avec la valeur de la première occurence de l'étiquette
'Initial'	initialise le descripteur de nombre qui vient d'être créé
'PVG' (;)	- en mode déclaration fait passer à mode instruction - en mode instruction effectue un branchement si un descripteur d'étiquette est présent au sommet de la pile ("A") dans tous les cas vide la pile d'évaluation
'Littéral'i	empile la valeur i
'ident' d	empile le descripteur de l'adresse "base de donnée"+d
'DPT' (:)	affecte au descripteur d'étiquette trouvé dans "A" le numéro de l'instruction suivante
'PLUS','MOINS','EG', 'SUP', 'ENFEG'	accèdent la valeur des opérandes, effectuent l'opération, mettent le résultat dans "B", vident "A" (lecture sur fichier si un des opérandes est de type entrée)
'Interro'(?)	si "B" contient la valeur "faux" vide "A"
'Affect'(→)	affecte la valeur contenue dans "B" à l'adresse précisée par le descripteur contenu dans "A" (écriture sur fichier si le descripteur dans "A" est une sortie)
'Fin'	effectue le branchement si une étiquette est en sommet de pile sinon fin de l'interprétation.

BIBLIOGRAPHIE

Projets et réalisations

[MS.63] MULLERY & SCHOUER & RICE - A problem oriented SYMBOL processor SJCC 1963.

[WE.67] H. WEBER - A microprogrammed Implementation of EULER on IBM 360/30 C.ACM N°9, 1967.

[HD.68] E.A. HAUCK & B.A. DENT - Burroughs B6500/B7500 Stack Mechanism, AFIPS, vol. 32, 1968.

[CR.69] CRE_ECH - Architecture of the B6500 COINS 69 - 3nd International Symposium on computer and information science - MIAMI 1969.

[CS.71] CHESLEY & SMITH - The hardware implemented high level machine language for SYMBOL - SJCC 1971.

[FO.73] B.E. FORBES, M.D. GREEN - An economical hull-scale multipurpose computer system - Hewlett-Packard Journal, January 1973.

[BU] Basic principles of the B65000 (Burroughs Corporation).

[HP] HP System / 3000 : System Description, Programming Manual (Hewlett-Packard).

[AB.70] ABRAHMS : An APL Machine, Stanford University, Computer Department, N° CS 70 158, 1970.

[DO.74] DOUSS_Y, MARTIN - Etude de définition d'une structure de petite machine, intégrant les primitives de base des systèmes et des langages évolués dans le domaine du temps réel - Rapport final, Contrat SESORI N° 73 041.

[BE.74] R.A. BELGRAD - An implementation of Blaise on the B1726 - TR 81, June 1974, University of New-York.

[GRI.74] GRIFFITHS - LL (1) grammar and analysis - Advanced Course in Compiler Construction, Spring Verlag, 1974.

[KN] D. KNUTH - The art of computer programming, vol. 1, 2, 3 - Addison Wesley Reading (Massachussetts).

[BER.74] BERTHAUD, GRIFFITHS - Incremental compilation and conversational interpretation - Annual Review in Automatic Programming n° 7, Perganon Press, 1974.

Publications de l'équipe

[FMS.73] R. FORTIER, J.P. MARTIN, J.P. SCHOELLKOPF - Etude d'une machine PASCAL - Rapport de D.E.A. Informatique, Université de Grenoble, Juin 1973.

[AF.74-1] F. ANCEAU, R. FORTIER, J.P. SCHOELLKOPF - Conception descendante des machines : application à une machine "PASCAL" - Journées AFCET machines orientées langage, machines orientées système. 2/3 mai 1974.

[SC.74-1] J.P. SCHOELLKOPF - A top-down approach of design and functionnal description of hardware - ACM German chapter, IEE German section, Workshop on CHDL's, DARMSTADT, 31/7 - 1/8/74.

[SC.74-2] J.P. SCHOELLKOPF - Microprogramming : a step of a top-down design methodology - 7th Annual Workshop on Microprogramming, SIGMICRO, PALO-ALTO, 29/9 - 1/10/74.

[AF.74-2] F. ANCEAU, R. FORTIER, J.P. SCHOELLKOPF - Conception descendante des machines informatiques : application à une machine PASCAL - Rapport final, Contrat SESORI n° 73 042, Novembre 1974.

[FO.75] R. FORTIER - Conception descendante d'une machine PASCAL - Journées sur l'implantation, le développement et les applications du langage PASCAL. Université de Nice, 5-6 juin 1975.

[BS.75-1] G. BAILLE, J.P. SCHOELLKOPF - Evaluation d'une expression post-fixée sur une file d'attente et réalisation d'une machine "pipe-line" de haut niveau - Séminaires de Programmation, Université de Grenoble, 6 juin 1975.

[BS.75-2] G. BAILLE, J.P. SCHOELLKOPF - Evaluation of "polish-form" expression on a FI-FO queue : a new approach towards the realization of a high-level "pipe-line" computer - SAGAMORE Computer Conference on Parallel Processing. August 19-22, 1975.

Langages

[WI.70] N. WIRTH - The Programming language PASCAL - 1970.

[WI.72] N. WIRTH - An axiomatic definition for PASCAL.

[WI.74] K. JENSEN and N. WIRTH - An user manual for PASCAL - April 1974.

[BO.72] (R.M.Woodward and S.G.Bond) Royal Rader Establishment. ALGOL 68-R.

Méthodologie

[IL.67] J.K. ILLIFE - Basic Machine Principles - AMER. ELSEVIER, New-York, 1968,

[KE.67] Mc KEEMAN - Language directed computer design - FICC 1967, pp. 413-417.

[BA.69] BARTON - Ideas for computer organization - COINS 69, 3nd International Symposium on computer and information science, Miami, 1969.

[GR.72] A. GREBERT - Space Programming Language Machine Architectures Study - SAMSO TR 72 117, May 1972.

[LE.72] J.A.N. LEE - Computer Semantics - Van Nostransd Reinhold Company, 1972.

[WO.72] D.B. WORTMAN - A study of language directed computer design - CSRG-20. VOF Torento, Dec. 1972.

[AN.73] F. ANCEAU - Systèmes hiérarchisés - Journées IRIA, St Pierre de Chartreuse, Novembre 1973.