Optimisation d'un compilateur incrémental et conversationnel de PL/I

Marie-Françoise Bruandet

THESE

présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

par

pour obtenir le grade de

Docteur de troisième cycle

Spécialité : INFORMATIQUE

Marie-Françoise BRUANDET

OPTIMISATION D'UN COMPILATEUR INCRÉMENTAL ET CONVERSATIONNEL DE PL/I

Thèse soutenue le 2 juillet 1976 devant la Commission d'Examen :

Président : N. GASTINEL

Rapporteur : M. GRIFFITHS

Examinateurs : M. BERTHAUD

Ph. JORRAND

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Michel SOUTIF : Président

Monsieur Gabriel CAU : Vice-Président

MEMBRES DU CORPS ENSEIGNANTS 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.	BERTRANDIAS Jean-Paul	Mathématiques Pures
	BEZES Henri	Pathologie Chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique Ophtalmologique
	BONNET-EYMARD Joseph	Clinique Gastro-entérologique
Mme	BONNIER Marie-Jeanne	Chimie Générale
MM.	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique Rhumatologique et Hydrologique
	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	Clinique de 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.	DEBELMAS Jacques	Géologie Générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie Minérale
	DESRE Pierre	Métallurgie

MM.	DESSAUX Georges	Physiologie Animale
	DODU Jacques	Mécanique Appliquée (I.U.T. A)
	DOLIQUE Jean-Michel	Physique des Plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	GAGNAIRE Didier	Chimie Physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse Numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	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
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie Générale
	LATURAZE Jean	Biochimie Pharmaceutique
	LAURENT Pierre-Jean	Mathématiques Appliquées
	LEDRU Jean	Clinique Médicale B
	LLIBOUTRY Louis	Géophysique
	LOISEAUX Pierre	Sciences Nucléaires
	LONGEQUEUE Jean-Pierre	Physique Nucléaires
	LOUP Jean	Géographie
Melle	LUTZ Elisabeth	Mathématiques Pures
	MALGRANGE Bernard	Mathématiques Pures
	BOUTET DE MONVEL Louis	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 (I.U.T. A)
	SOUTIF Michel	Physique Générale
	TANCHE Maurice	Physiologie

MM.	TRAYNARD Philippe	Chimie Générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique Nucléaire
	VAUQUOIS Bernard	Calcul Electronique
Mme	VERAIN Alice	Pharmacie Galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique Nucléaire Théorique

PROFESSEURS ASSOCIES

MM.	CLARK Gilbert	Spectrométrie Physique
	CRABBE Pierre	CERMO
	ENGLMAN Robert	Spectrométrie Physique
	HOLTZBERG Frédéric	Basses Températures
	ROST Ernest	Sciences Nucléaires

PROFESSEURS SANS CHAIRE

Melle	AGNIUS-DELORD Claudine	Physique Pharmaceutique
	ALARY Josette	Chimie Analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques Appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. A)
	BUISSON René	Physique (I.U.T. A)
	CONTE René	Physique (I.U.T. A)
	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 Médecine Préventive
	IDELMAN Simon	Physiologie Animale
	JOLY Jean-René	Mathématiques Pures
	JULLIEN Pierre	Mathématiques Appliquées
Mme	KAHANE Josette	Physique
MM.	KUHN Gérard	Physique (I.U.T. A)
	LE ROY Philippe	Mécanique (I.U.T. A)
	LUU DUC Cuong	Chimie Organique
	MAYNARD Roger	Physique du Solide
	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et Minéralogie
	PSISTER Jean-Claude	Physique du Solide
Melle	PIERY Yvette	Physiologie Animale
MM.	RAYNAUD Hervé	M.I.A.G.
	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

MM.	SARRAZIN Roger	Anatomie et Chirurgie
	SARROT-REYNAUD Jean	Géologie
	SIROT Louis	Chirurgie Générale
Mme	SOUTIF Jeanne	Physique Générale
MM.	STREGLITZ Paul	Anesthésiologie
	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 (I.U.T. A)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro chirurgie
	BARJOLLE Michel	MIAG
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (C.U.S.)
MM.	BRODEAU François	Mathématiques (I.U.T. B)
	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
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	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 (C.U.S.)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine Légale
	FAURE Gilbert	Urologie
	FONTAINE Jean-Marc	Mathématiques Pures
	GAUTIER Robert	Chirurgie Générale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. A)
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KOLODIE Lucien	Hématologie
	KRAKOWIAK Sacha	Mathématiques Appliquées
	LE NOC Pierre	Bactériologie-virologie
	LEROY Philippe	I.U.T. A
	MACHE Régis	Physiologie Végétale
	MAGNIN Robert	Hygiène et Médecine Préventive
	MALLION Jean-Michel	Médecine du Travail
	MARECHAL Jean	Mécanique (I.U.T. A)
	MARTIN-BOUYER Michel	Chimie (C.U.S.)

M.	MICHOULIER Jean	Physique (I.U.T. A)
Mme	MINIER Colette	Physique (I.U.T. A)
MM.	NEGRE Robert	Mécanique (I.U.T. A)
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T.A)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B)
	PEFFEN René	Métallurgie (I.U.T. A)
	PERRET Jean	Neurologie
	PERRIER Guy	Géophysique - Glaciologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine Interne
	RACINET Claude	Gynécologie et Obstétrique
	RAMBAUD André	Hygiène et Hydrologie
	RAMBAUD Pierre	Pédiatrie
Mme	RENAUDET Jacqueline	Bactériologie
MM.	ROBERT Jean-Bernard	Chimie-Physique
	ROMIER Guy	Mathématiques (I.U.T. B)
	SHOM Jean-Claude	Chimie générale
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

MAITRE DE CONFERENCES ASSOCIES

M.	COLE Antony	Sciences Nucléaires

CHARGE DE FONCTIONS DE MAITRE DE CONFERENCES

M.	JUNIEN-LAVILLAVROY Paul	O.R.L.

Fait à SAINT MARTIN D'HERES,
DECEMBRE 1975.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : M. NEEL Louis

Vice-Présidents : M. BENOIT Jean

M. BONNETAIN Lucien

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BLOCH Daniel	Physique du solide
	BONNETAIN Lucien	Chimie Minérale
	BONNIER Etienne	Electrochimie et Electrométallurgie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	COUMES André	Radioélectricité
	FELICI Noël	Electrostatique
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie-Physique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	POLOUJADOFF Michel	Electrotechnique
	SILBERT Robert	Mécanique des Fluides

PROFESSEURS ASSOCIES

MM.	RUPPERSBERG Albert, Henner	Chimie
	ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BOUVARD Maurice	Génie Mécanique
	COHEN Joseph	Electrotechnique
	DURAND Francis	Métallurgie
	FOULARD Claude	Automatique
	LACOUME Jean-Louis	Géophysique
	LANCIA Roland	Electronique
	VEILLON Gérard	Informatique Fondamentale & Appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Mathématiques Appliquées
	BOUDOURIS Georges	Radioélectricité
	CHARTIER Germain	Electronique
	GUYOT Pierre	Chimie Minérale
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du solide
	MORET Roger	Electrotechnique Nucléaire
	PIERRARD Jean-Marie	Hydraulique
	ROBERT François	Analyse Numérique
	SABONNADIERE Jean-Claude	Informatique Fondamentale & Appliquée
Mme	SAUCIER Gabrièle	Informatique Fondamentale & Appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan	Automatique

CHERCHEURS DU C.N.R.S. (Directeurs et Maîtres de Recherche)

MM.	FRUCHART Robert	Directeur de Recherche
	ANSARA Ibrahim	Maître de Recherche
	CARRE René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Je remercie

Monsieur le Professeur N. GASTINEL, qui a bien voulu s'intéresser à ce travail et accepter de présider mon jury de thèse.

Monsieur M. GRIFFITHS, qui m'a accueillie dans son équipe et a accepté de diriger cette thèse. J'ai apprécié la liberté et l'initiative qu'il m'a laissées et ses conseils, lors de la rédaction de cette thèse, m'ont été très utiles.

Monsieur M. BERTHAUD, dont l'aide constante a été indispensable à la réalisation de ce travail, et je lui suis très reconnaissante de m'avoir fait bénéficier, avec beaucoup de gentillesse, de sa grande expérience.

Monsieur Ph. JORRAND, qui m'a fait l'honneur d'être membre du jury.

Ce travail a aussi bénéficié, dans les périodes où les erreurs de programmation faisaient mon désespoir, de l'aide de mes collègues de Laboratoire, et en particulier de celle de Michel DELAUNAY.

Je remercie enfin Mademoiselle E. NAUDIN, Madame C. PUECH et Messieurs ANGUILLE, IGLESIAS, LABORIE, MOLLIET, MOUNET qui ont fait tout leur possible pour réduire les délais de réalisation de cette thèse.

PREMIERE PARTIE

DEUXIEME PARTIE

PREMIERE PARTIE

CHAPITRE I

BREVE DESCRIPTION DU COMPILATEUR INCREMENTAL ET CONVERSATIONNEL DE PL/I

I - INTRODUCTION

Le compilateur incrémental et conversationnel de PL/1 a été réalisé par un groupe mixte 'Université de Grenoble, Centre Scientifique I.B.M. France'.

La structure de ce compilateur et les principes adoptés pour résoudre les différents problèmes ont été exposés par MM. M. GRIFFITHS et M. BERTHAUD [BERT-GRIFF] ; ont également travaillé sur ce projet Mme M. CHABRE et MM. M. PELTIER et Ph. JORRAND.

D'autres projets ont été réalisés pour d'autres langages par M. D. CLAUZEL et Mme V. BAJAR [BAJ] pour le langage FORTRAN, et MM. Y. CUNIN, B. BRETAGNOLLE, B. MAILLOT PILARD [CUN] [BRETA] pour le langage ALGOL 60.

Pour mettre en évidence les difficultés et pour qu'apparaissent plus clairement les solutions proposées ainsi que la nécessité d'une optimisation nous rappelerons les principaux éléments de ce compilateur.

Ce type de compilateur offre à l'utilisateur des facilités de modifications au moment de l'édition du programme, ainsi que des possibilités d'interaction au moment de l'exécution du programme. Pour avoir le maximum de possibilités à l'exécution, contrairement à un compilateur classique, ce compilateur ne produit pas à partir du programme source un code machine (qui peut être directement exécuté par l'ordinateur.

On considère un incrément du programme source à la fois (en général)

une instruction) et toute l'information nécessaire pour générer le code machine n'est pas disponible. Le programme est donc transformé en un code intermédiaire ou pseudo-code.

Ce pseudo-code est interprété. L'interpréteur lit le pseudo-code et exécute les actions indiquées par celui-ci.

L'objectif de ce compilateur incrémental et conversationnel est de permettre à l'utilisateur un "dialogue avec son programme".

Ce dialogue peut avoir lieu aussi bien au moment où il crée son programme à la console, c'est-à-dire à l'édition, que pendant l'exécution (partielle ou totale) de celui-ci. L'exécution peut d'ailleurs être demandée par l'utilisateur, même s'il existe des erreurs syntaxiques dans le programme.

Il peut aussi arrêter l'exécution et introduire une ou plusieurs instructions consécutives, les exécuter immédiatement sans les intégrer dans le programme source.

L'utilisateur travaille sur une partie de programme, que l'on appelle segment. Un segment est simplement une suite d'instructions. C'est sur un segment, et un seul à la fois, que l'utilisateur a la possibilité d'agir : c'est-à-dire le créer, le modifier ou l'exécuter.

Pour mémoire, nous rappelerons quelques autres caractéristiques du compilateur ; en particulier, un segment est limité à 255 incréments et le nombre d'identificateurs différents ne doit pas dépasser 128.

D'autre part, ce compilateur travaille dans l'environnement du système conversationnel LCMS [BELLINO] ; ce système a entre autres, comme tâche, d'assurer les communications avec le terminal, de gérer la mémoire, d'allouer des zones de travail à chaque terminal, et aussi de donner le contrôle, successivement à chacun des terminaux utilisés.

II - STRUCTURE DU COMPILATEUR

Le compilateur est formé de deux phases successives : un générateur et un interpréteur.

2.1 - Le générateur

L'unité de travail du générateur est l'incrément, ce qui signifie que le générateur analyse syntaxiquement un incrément indépendamment du contexte dans lequel se trouve cet incrément.

Le générateur remplit, d'autre part, les quatre fonctions suivantes :

- numérotation des incréments
- création d'un dictionnaire de commandes
- création d'une table d'identificateurs
- génération d'un pseudo-code.

. Numérotation des incréments

Le générateur numérote les incréments, et le numéro du nouvel incrément est imprimé (au début de la ligne), avant que l'utilisateur ne tape cet incrément au terminal).

Lorsqu'une erreur "locale" est détectée dans un incrément, il y a impression d'un message d'erreur. Dans le cas d'une erreur, le générateur ignore cet incrément et ré-imprime, au début de la ligne suivante, le même numéro d'inrément comme nous le voyons dans l'exemple suivant.

```
SEGMENT essai;
 NEW SEGMENT:
001 E :BEGIN;
    $
E-SYNTAX ERROR
001 e:BEGIN;
002 DCL c FIXED BIN;
003 DCL m FIXED DEC INITIAL (2.98);
004 DCL b;DCL n;


006 LIST;          ------------------------------------> COMMANDE DEMANDANT L'IMPRESSION DU
 ***                                                     PROGRAMME EDITE
 001      e:BEGIN;
 002      DECLARE c BINARY FIXED(15,0);
 003      DECLARE m DECIMAL FIXED(5,0) INITIAL(2.98);
 004      DECLARE b;
 005      DECLARE n;
 ***

006 AFTER 1;       ------------------------------------> INDIQUE QUE L'ON VA EFFECTUER DES
006 DCL z;                                               MODIFICATIONS APRES L'INCREMENT 1
007 LIST;
 ***
 001      e:BEGIN;
 006      DECLARE z;
 002      DECLARE c BINARY FIXED(15,0);
 003      DECLARE m DECIMAL FIXED(5,0) INITIAL(2.98);
 004      DECLARE b;
 005      DECLARE n;
 ***

007 SKIP   2;      ------------------------------------> INDIQUE QUE L'ON SUPPRIME L'INCREMENT 2.
007 LIST;
 ***

 001      e:BEGIN;
 006      DECLARE z;
 003      DECLARE m DECIMAL FIXED(5,0) INITIAL(2.98);
 004      DECLARE b;
 005      DECLARE n;
 ***
007 AFTER 2;
E-INCREMENT NOT FOUND
007 QUIT;
 ***QUIT SEGMENT essai
```

. Dictionnaire de commande .

L'analyse de l'incrément permet d'en connaître le type, instruction BEGIN, instruction d'affectation, etc....

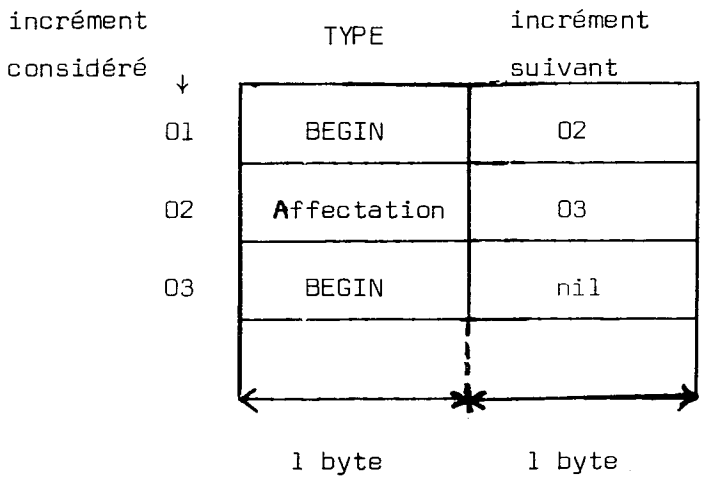
Cette information est conservée pour chaque incrément dans le dictionnaire de commande. A chaque incrément est associé le numéro de l'incrément suivant (ordre lexicographique du programme). L'ordre des instructions peut donc être modifié sans entraîner la reconstruction complète du dictionnaire de commande.

Exemple :

. Programme

```
01      BEGIN ;
02      x = 2 ;
03      BEGIN ;
04      .....
```

. Dictionnaire de commande



L'utilisateur peut modifier l'ordre lexicographique de son programme grâce aux commandes d'édition.

Si, dans le programme précédent, à l'incrément 04 on écrit :

```
04      AFTER   1 ;
04      a =     10 ;
05      .....
```

la commande AFTER 1 signifie que l'incrément 4 suivra désormais l'incrément 1 ; en fait, il sera ainsi inséré après l'incrément 1 ainsi que les incréments suivants.

Ce qui donnera pour le dictionnaire de commande :

	Type	incrément suivant
01	BEGIN [12]*	04
02	Affectation [17]	03
03	BEGIN [12]	05
04	Affectation [17]	02

* Les nombres entiers ainsi donnés sont la représentation dans le dictionnaire de commande du type des incréments.

. Création d'une table d'identificateurs.

Chaque identificateur est remplacé par un index, Deux occurrences d'un d'un même identificateur, même si elles correspondent à des variables différentes, ont le même index.

. Pseudo-code

Le pseudo-code, ou code intermédiaire, fourni par le générateur est organisé en blocs de 1 k octets, éventuellement placés en mémoire auxiliaire.

On accède au pseudo-code créé pour un incrément à l'aide du numéro de cet incrément. Ce code intermédiaire doit être compact et facile à interpréter (par exemple, les expressions sont écrites dans ce code en notation postfixée). De plus, ce pseudo-code est symétrique à quelques détails lexicographiques près ; on peut, si on le désire, recréer le programme source [CUN] . Une description du pseudo-code est donnée en annexe de ce chapitre.

2.2-L'interpréteur

L'interpréteur est constitué d'un ensemble de fonctions sémantiques qui sont activées par la commande "EXECUTE nom-de-segment ;".

Les principaux éléments utilisés par l'interpréteur sont le dictionnaire de commande et le pseudo-code.

Une table des symboles fournit les fonctions d'accès à la pile d'exécution en fonction de l'index de l'identificateur.

La pile contiendra les attributs concernant les types des variables ainsi que leurs valeurs. Du fait de la structure de blocs du langage, il peut exister, pour un même identificateur, plusieurs déclarations (dans des blocs différents).

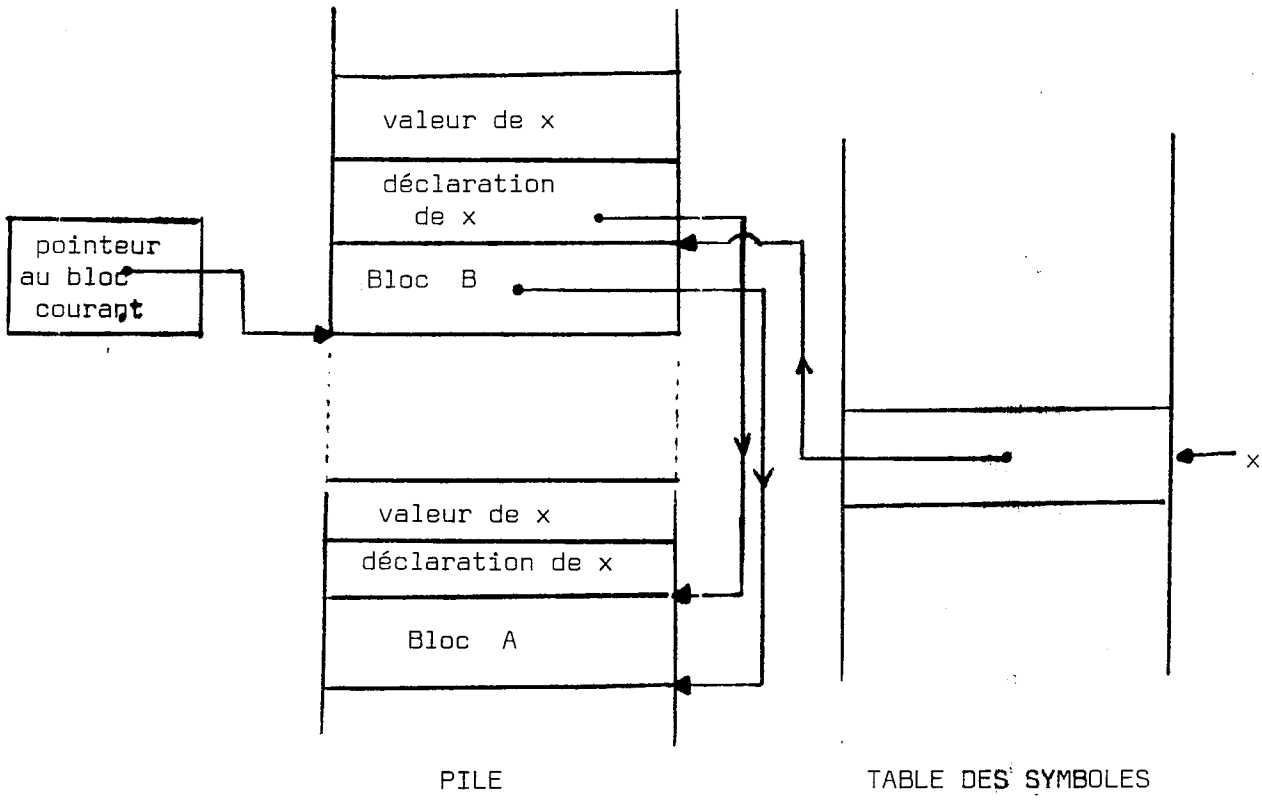
Pour un même nom d'identificateur, les déclarations sont chaînées : la représentation de la dernière déclaration traitée contient un pointeur à la précédente déclaration. La table des symboles permet, en fonction de l'index de l'identificateur, d'accéder à la dernière déclaration valide traitée. Ce chaînage sera utilisé pour la mise à jour de la table des symboles à la fin de l'exécution d'un bloc et pour la libération de la mémoire allouée aux variables automatiques de ce bloc.

De plus, la pile contient des informations donnant le type du bloc (BEGIN ou PROCEDURE) et le chaînage entre les blocs accessibles à un instant donné (chaîne dynamique des blocs).

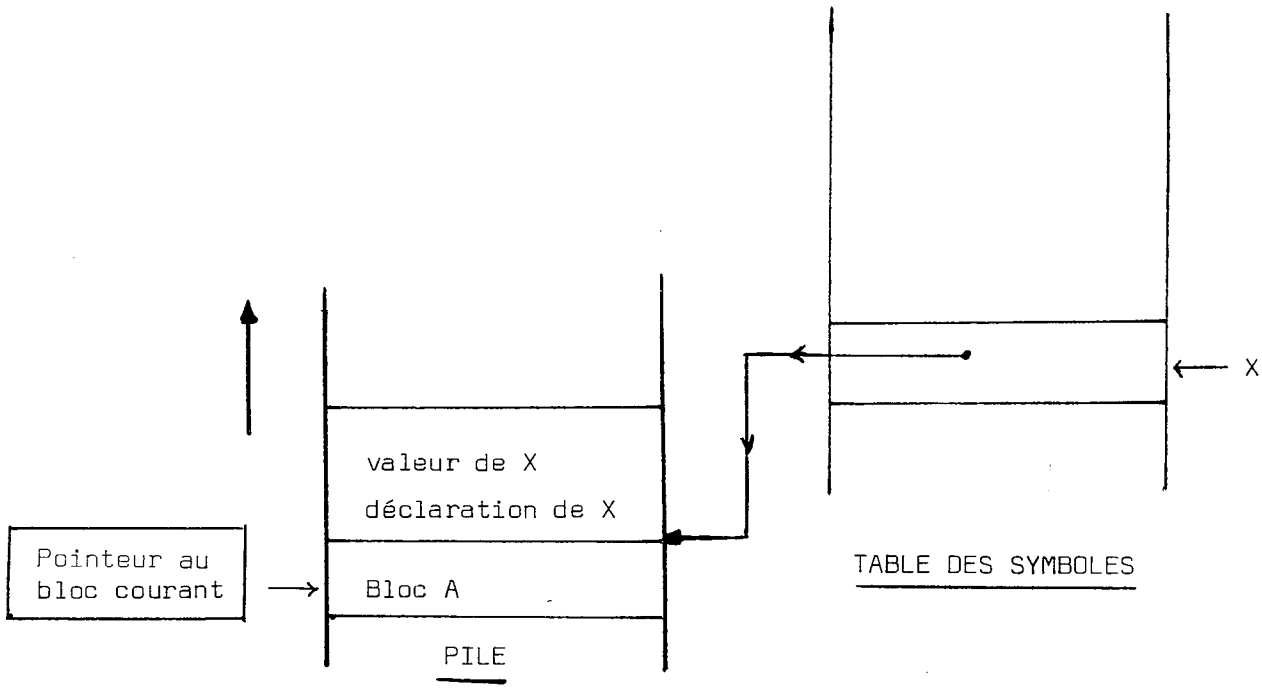
Exemple :

```
01   A : BEGIN ;
02   DCL  X ;
03   B : BEGIN ;
04   DCL  X  CHAR  (9) ;
05   X : 'CHARACTER' ;
06   END B ;
07   .....
```

A l'exécution de l'incrément 5, l'état de la pile et de la table des symboles sera



A l'exécution de l'incrément 7, la pile ne contiendra plus aucune information pour le bloc B puisque celui-ci est inactif. L'état de la pile et de la table des symboles est le suivant :



Le dictionnaire de commande permet à l'interpréteur de déterminer l'ordre dans lequel les incréments doivent être exécutés.

L'interpréteur se charge donc :

- de l'adressage et de l'allocation de mémoire aux variables.
- des tests de compatibilité et des conversions des opérandes dans le cas d'expressions
- de l'exécution et de l'enchaînement des instructions.

Le travail de l'interpréteur est donc important par rapport à celui du générateur.

2.3-Niveau du langage implémenté

Le niveau du sous-ensemble de PL/I ainsi implémenté correspond au niveau du langage ALGOL 60 avec des éléments supplémentaires : notamment l'existence des variables structurées, la manipulation de variables non numériques (chaînes), la possibilité d'initialiser les variables lors de leurs déclarations.

Les entrées-sorties admises sont limitées au terminal (instructions GET et PUT avec la seule option LIST).

Les éléments admis sont les suivants :

- déclarations de variables arithmétique de classe DECIMAL, BINARY, de représentation FIXED ou FLOAT
- déclarations de chaînes de caractères ou de BIT
- déclarations de tableaux
- déclarations de structure
- les procédures
- l'instruction DO
- l'instruction IF
- l'instruction END et la fermeture multiple de blocs ou de groupes
- l'instruction GOTO
- l'instruction d'affectation

Ne sont pas pris en compte

- les ON-conditions
- les "multi-tâches"
- les entrées-sorties de type RECORD
- les données AREA et OFFSET

Les programmes ont été écrits en assembleur IBM/360 avec l'aide d'un système de macros [GRIFF.PELT] , [PELTIER] .

. Annexe au chapitre I

Le pseudo-code est donné sous forme d'une grammaire avec les conventions suivantes :

- Les caractères entre crochets sont indiqués dans le dictionnaire de contrôle
- Les caractères entre parenthèses sont dans le pseudo-code
- Les caractères entourés ni de parenthèses ni de crochets sont les éléments de la grammaire
- Une astérisque (*) indique une répétition de 1 à n fois.

Une nouvelle ligne indique une alternative, certains développements n'ont pas été donnés tels que "nombre de dim" qui est un entier.

La grammaire est donnée de façon indicative.

Incrément → [type-de-l'incrément]
[étiquette+type-de-l'incrément] liste-d'étiquettes

Liste-d'étiquettes → Var (fin-de-la-partie-étiquette)
Var liste-d'étiquettes

Type-de-l'incrément → BEGIN
END
Déclaration
IF
ELSE
PROCEDURE
DO
Affectation
PUT LIST
GET LIST
CALL
RETURN

BEGIN → [12]
END → [7]
Endmult

Endmult → [8] (index)

Déclaration → [1] simpledec initial
[1] Arraydec initial
[1] structdec initial

Simpledec → (Index) Liste-de-type
Arraydec → borne-de-tableau
borne-de-tableau Arraydec

borne-de-tableau → (Index 5 nombre-de-dim) liste-de-type Ex^*

Ex → $Elément^*$ (9)
Element → Const
Var
Opérateur
crochet

Const → (5) liste-de-type (valeur)

Var → Id simple (Index)
Id indicé (Index nombre-d'élément) Ex^*
Id qual Id-liste

Idsimple → (1)
Idqual → (2)
Idindicé → (3)

Id-liste → (nombre-d'identificateurs $index^*$)

Structdec → (Index 4) Elément-de-structure

Element-de-structure → liste-de-structure
liste-de-structure (virgule) élément-de-structure

Liste-de-structure → (Index niveau) liste-de-type
(Index niveau nombre-de-successeurs Index*)

Liste-de-type → Arithmétique-liste
Stringlist

Arithmétique-liste → (Arithmetic coded form 1) type (prec scalefactor longueur)

type → base Echelle mode

base → (BINARY)
(DECIMAL)

Echelle → (FIXED)
(FLOAT)

mode → (REAL)
(COMPLEX)

Stringlist → (4) type-de-longueur type-de-chaîne (nombre d'éléments char*)

Type-de-longueur → (VARYING)
(FIXED)

Type-de-chaîne → (BIT) caractéristique
(CHAR)

Caractéristique → (UNALIGNED)
(ALIGNED)

IF → ⌈16⌉ Ex

ELSE → ⌊15⌋

GOTO → [18]

PROCEDURE → [13] Spécification

Spécification → [8] (nombre-de-paramètre)liste-de nom-de-point-d'entrée(Index*)

Liste-de-nom-de-point → (Index)liste-de-type fin-de-la-partie-entryname
(Index)liste-de-type list-de-nom-de-point d'entrée

DO → [9]
[10] DOwhile
[11] DOitération

DOitération → EX TO EX
EX BY EX
EX BY EX TO EX

TO → (1)

BY → (2)

DOwhile → WHILE EX
DOitération WHILE EX

WHILE → (4)

Affectation → [17] Var* egal (nombre-d'élément-en-partie-gauche) EX

Egal → [11]

PUTLIST → [23] Id-liste

GETLIST → [22] Id-list

CALL → [19] Var

RETURN → [20]

INITIAL → (253 valeur)

CHAPITRE II

NECESSITE D'UNE OPTIMISATION

I-INTRODUCTION

La structure décrite pour le compilateur incrémental implique un lourd travail à l'interprétation soit une vitesse d'exécution relativement faible.

En effet, c'est à l'exécution (travail de l'interpréteur) que l'on effectue la plupart des tâches : reconnaissance des noms, conversions, allocation de mémoires et exécution des instructions. La "compilation" (tâche du générateur) est réduite au minimum du fait du traitement incrément par incrément.

Les performances du compilateur ont été partiellement mesurées et on peut estimer que les temps d'interprétation sont 60 à 150 fois plus élevés que les temps d'exécution des mêmes programmes traités par le compilateur PL/I (F) sur IBM 360 [BERTHAUD].

Notre critère d'optimisation pour ce compilateur sera donc le critère temps. Ce critère nous paraissant très important, nous ne nous sommes pas occupé du critère espace de mémoire nécessaire au pseudo-code. Nous utilisons un espace de travail de 4096 bytes et le bloc de pseudo-code est mis sur une mémoire auxiliaire lorsqu'il est rempli.

II - SOLUTIONS PROPOSEES

Le nombre de tâches assurées par l'interpréteur dépend essentiellement des informations fournies par le générateur au moyen du pseudo-code et du dictionnaire de commande.

Une des solutions en vue d'accélérer l'interprétation est d'interpréter un pseudo-code plus complet. Les fonctions du générateur sont alors augmentées, et il se charge, non seulement de l'analyse syntaxique, mais aussi d'une partie de l'analyse sémantique statique (autrement dit certaines liaisons entre les différents incréments sont faites à la génération : instructions IF-THEN, association BEGIN-END, etc...).

Dans le compilateur, tel qu'il existe, on ne considère qu'un incrément ; chaque modification (suppression ou création d'un incrément) ne modifie que le pseudo-code et le dictionnaire de commande relatifs à cet incrément.

Considérons, non plus un incrément, mais un groupe d'incréments, par exemple un bloc. Effectuons les liaisons existant entre les incréments de ce groupe (en particulier l'association "utilisation-déclaration" des variables). On peut alors définir, pour le générateur, une unité de travail plus grande : le groupe d'incréments choisi. Le travail de l'interpréteur est ainsi allégé ; par contre, la modification d'un incrément, dans ce groupe peut obliger à recompiler tout le groupe en tenant compte de ce nouvel incrément.

Par exemple, si le groupe choisi est un bloc, la modification d'une déclaration oblige le générateur à réexaminer toutes les expressions du bloc, et à recompiler tous les incréments du bloc.

Nous sommes alors obligés, soit de recompiler systématiquement les incréments du bloc, soit d'analyser les incréments modifiés, afin d'examiner s'ils ont une interaction sur les autres incréments du bloc.

Il y a certaines modifications qui peuvent entraîner un temps de recompilation important. Par exemple, si le segment est constitué d'un bloc, la modification d'une déclaration de ce bloc entraîne la recompilation du segment complet. Une organisation de ce type a été réalisée pour le compilateur CPL1 [CPL1] ; de plus, pour que l'association utilisation-déclarations des variables soit faite à la compilation, les déclarations doivent être mises lexicalement avant leur utilisation (sauf si le compilateur est en plusieurs passages).

Pour conserver à l'utilisateur toutes les possibilités d'interaction au moment de la création de son programme, nous ne modifierons pas les fonctions du générateur. Ce dernier est d'ailleurs satisfaisant au point de vue rapidité. Notre effort d'optimisation s'est donc porté sur l'interpréteur.

Nous nous proposons de modifier l'interpréteur en séparant les deux notions : l'association "utilisation-déclaration" des variables et l'allocation de la mémoire aux variables.

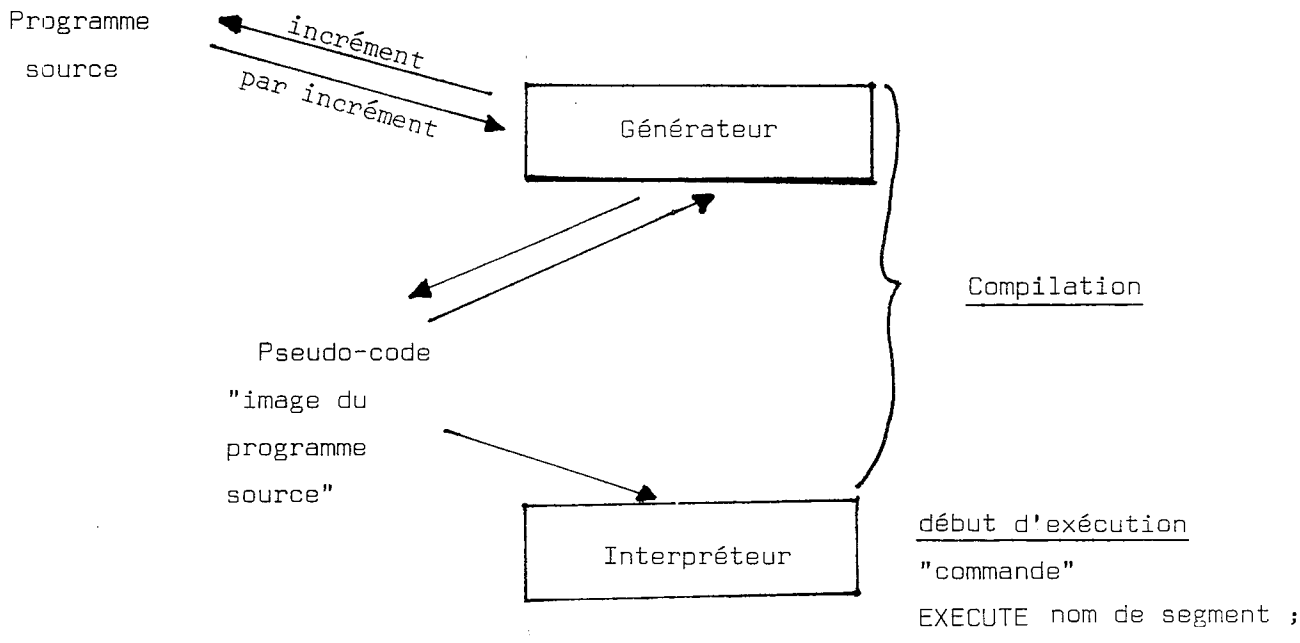
L'association des noms sera faite dans une première phase, appelée phase de pré-interprétation. Cette phase se situera après le générateur, lorsque l'utilisateur demande l'exécution de son programme. Au cours de cette phase, on effectuera certaines liaisons entre les incréments, en particulier l'association "utilisation-déclaration" des variables, dont les résultats seront transmis à la phase d'interprétation proprement dite, par l'intermédiaire d'un nouveau pseudo-code.

L'ancien pseudo-code sera conservé comme image du programme source.

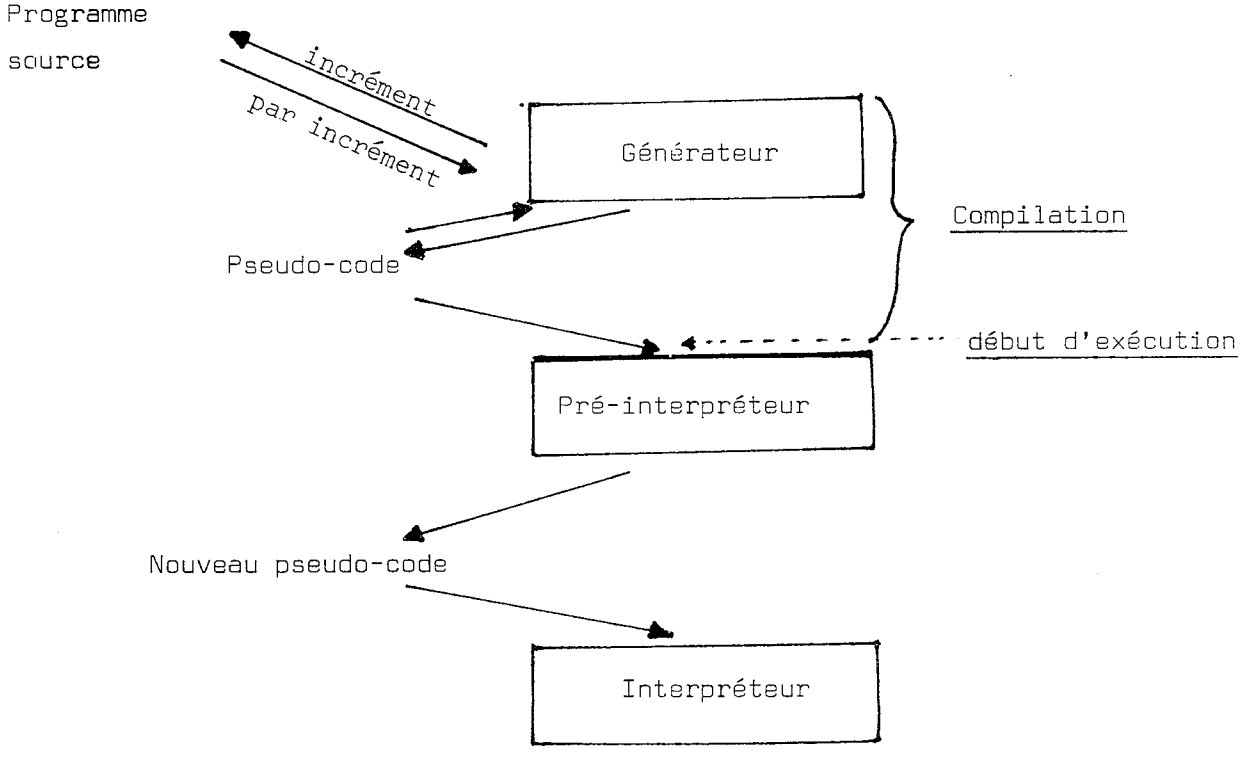
La phase d'interprétation sera chargée de l'enchaînement des incréments de l'allocation de mémoire aux variables et de l'exécution des incréments en utilisant le nouveau pseudo-code.

L'enchaînement de ces différentes phases est représenté par les schémas suivants :

Compilateur incrémental et conversationnel



Compilateur incrémental et conversationnel avec pré-interpréteur



Nous pourrions envisager également une optimisation au niveau de l'incrément [BERTHAUD].

Chaque fois qu'un incrément nouveau est exécuté, nous pourrions conserver les instructions machines nécessaires à son exécution dans le nouveau pseudo-code. C'est-à-dire que pour chaque incrément on conserve, après la première exécution, son image en langage machine (programme objet).

Cela suppose que l'on puisse connaître quel est l'état de l'incrément ainsi que ceux qui sont déjà été exécutés.

Cette étape pourrait être envisagée après la première optimisation que nous proposons.

III - PROBLEMES A RESOUDRE

Du fait de la définition incrémentale du compilateur, l'interpréteur exécute un certain nombre d'opérations. Passons en revue et analysons les principales opérations responsables de l'augmentation des temps d'exécution.

- Les déclarations sont évaluées à chaque exécution d'un bloc. Cette opération, du fait que les déclarations des identificateurs peuvent être situées après leurs utilisations, oblige à analyser tous les incréments de ce bloc.

- L'association entre les références à un identificateur et sa déclaration n'est faite qu'à l'interprétation. Ce qui implique la recherche de la déclaration d'un identificateur (apparaissant dans une instruction donnée) ; l'analyse des conversions à effectuer sur les valeurs est faite chaque fois que l'instruction est exécutée.

Par exemple, l'interprétation de l'instruction

x = 1 ;

(l'identificateur x ayant été déclaré FIXED BIN) provoquera la conversion de la constante 1 à la représentation de x, toutes les fois que l'instruction d'affectation sera exécutée.

Nous nous proposons de traiter les éléments précédents, avant l'exécution, dans la phase de pré-interprétation.

Nous conserverons le pseudo-code fourni par le générateur et créerons un nouveau pseudo-code (introduisant certaines liaisons entre les incréments, en particulier entre utilisations et déclarations des variables).

Nous distinguerons deux niveaux d'optimisation : une optimisation au niveau de la structure du segment (à l'aide du dictionnaire de commande), une autre au niveau des incréments (à l'aide d'un dictionnaire d'attributs).

Le premier niveau d'optimisation a été réalisé par [BERTHAUD].

Nous effectuons le processus d'identification des variables dans la phase de pré-interprétation à l'aide du dictionnaire d'attributs, dans lequel nous conserverons les attributs des variables. L'accès à ce dictionnaire se fera au moyen d'une table des symboles.

Pour l'allocation de la mémoire aux variables AUTOMATIC, nous utiliserons une pile (au moment de l'exécution). Une table des blocs nous permettra de savoir quels sont les blocs actifs à un instant donné.

Nous décrirons dans les chapitres suivants le rôle et la structure des éléments que nous créons (nouveau pseudo-code, dictionnaire d'attributs, table des symboles, pile, table des blocs).

De plus, du fait que l'on analyse les incréments avant l'exécution, nous pourrons traiter d'autres problèmes dans cette phase.

En particulier, le traitement des déclarations peut être long et complexe, puisque les déclarations peuvent contenir des expressions. Ces expressions définissent les longueurs des chaînes, les bornes des tableaux, les facteurs d'itérations lors d'initialisation avec l'attribut INITIAL. Les déclarations peuvent aussi faire intervenir d'autres identificateurs lorsqu'elles contiennent les attributs LIKE, DEFINED et BASED (pour les deux derniers attributs, leur évaluation ne se fait que lors de l'utilisation). Lorsque les identificateurs, intervenant dans ces déclarations, sont déclarés dans le même bloc (que les déclarations que l'on veut traiter), un problème se pose : dans quel ordre faut-il traiter ces déclarations ? Nous déterminerons, lorsque cela est possible, l'ordre d'exécution des déclarations.

Nous pourrons traiter les variables STATIC ; il est possible, avant l'interprétation, plus précisément avant l'allocation de mémoire aux variables AUTOMATIC, de connaître le nombre de ces variables ainsi que la place qui leur est nécessaire, et la réserver en début d'exécution.

Nous pourrons également traiter les variables implicites. Nous effectuons dans cette phase l'identification des variables, ce qui nous permet de déceler les variables pour lesquelles il n'existe pas de déclarations explicites. Nous effectuons, pour cet identificateur, une déclaration implicite en utilisant les attributs par défaut.

DEUXIEME PARTIE

CHAPITRE I

DETECTION DE LA STRUCTURE DU PROGRAMME

I - RECHERCHE DES LIAISONS ENTRE INCREMENTS

Le dictionnaire de commande décrit l'enchaînement des incréments dans le programme source. L'ordre d'exécution des incréments dépend de leur nature et, éventuellement, des résultats de l'exécution d'autres incréments. Nous pouvons avoir plusieurs exécutions d'une même instruction, en particulier d'une instruction BEGIN ou PROCEDURE.

Pour de telles instructions, on procède à l'ouverture d'un nouveau bloc, et on traite les déclarations des variables locales à ce bloc. Pour trouver toutes les déclarations explicites d'un bloc, on est obligé de parcourir le dictionnaire de commande pour examiner les incréments du bloc. On saute les incréments des blocs imbriqués, et pour cela, on analyse le dictionnaire de commande et le pseudo-code dans le cas de END multiples.

Pour simplifier et faciliter ce travail [BERTHAUD] a réalisé,à l'intérieur du dictionnaire de commande, une liaison entre les BEGIN-END, PROCEDURE-END, IF-ELSE et DO-END, en y introduisant un renseignement supplémentaire : parmi les informations concernant l'incrément correspondant au début de bloc ou du groupe DØ, on trouvera le numéro de l'incrément END correspondant (et inversement). Ce qui permet de résoudre dès cette phase le problème des END multiples.

Montrons, sur un exemple, ce que devient le dictionnaire de commande.

```
01      BEGIN ;
02      DCL A ;
03        A=1 ;
04      E : BEGIN ;
05      DCL X ;
06        X=A ;
07      BEGIN ;
08      X=A+X ;
09      END E ;
10      END ;
```

Le dictionnaire de commande est dans l'état suivant :

	incrément suivant	Type de l'incrément	Chaînage
	PNT DICT	TYPE DICT	PREV DICT
01	02	BEGIN [12]	10
02	03	DCL [1]	00
03	04	Affectation [17]	00
04	05	Etiquette + Begin [140]	09
05	06	DCL [1]	00
06	07	BEGIN [12]	09
07	08	Affectation [17]	00
08	09	Endmultiple [8]	04
09	10	END [7]	01
10	00	00	00

II - DETECTION DE LA CHAINE STATIQUE DES BLOCS. CREATION DE LA TABLE DES BLOCS

Après l'édition de son programme (une fois le travail du générateur terminé), le programmeur peut en demander l'exécution.

Le premier travail de la phase de pré-interprétation est alors de créer une table des blocs. Cette table permet de déterminer, à un instant donné, quelle est la déclaration valide pour un identificateur. Cette table devra rendre compte de l'enchaînement statique des blocs ; en d'autres termes, elle permet de savoir quels sont les blocs englobant le bloc dans lequel on se trouve.

Pour cela les blocs BEGIN et PROCEDURE sont numérotés dans l'ordre où ils apparaissent dans le programme. Ces numéros sont rangés dans une zone supplémentaire du dictionnaire de commande à l'incrément correspondant à l'instruction BEGIN ou PROCEDURE.

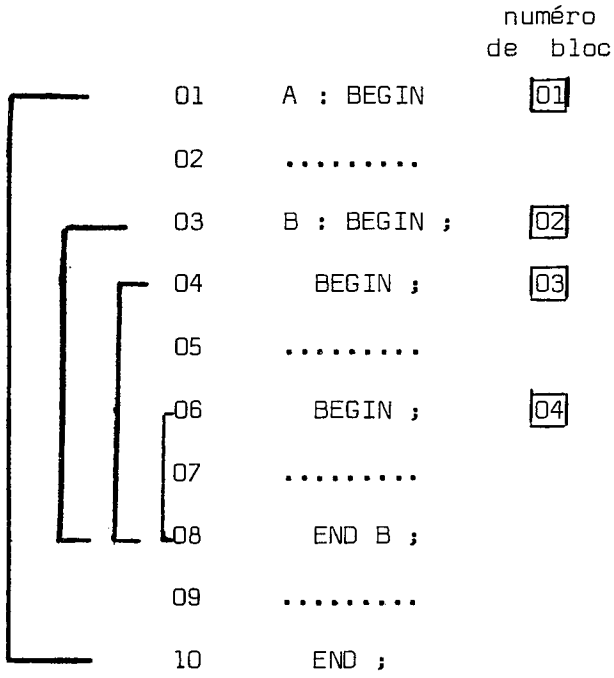
Pour créer la table des blocs, nous utilisons le chaînage des BEGIN-END et des PROC-END effectué dans la phase précédente. Chaque fois qu'un incrément BEGIN ou PROCEDURE est rencontré, on lui affecte un numéro ; la numérotation est faite simplement de 1 en 1.

Pour pouvoir traiter le cas des END multiples, on utilise une pile. Chaque entrée comprend le numéro du bloc traité ainsi que le numéro d'incrément du END correspondant (obtenu par le chaînage donné dans le dictionnaire de commande lors du premier niveau d'optimisation).

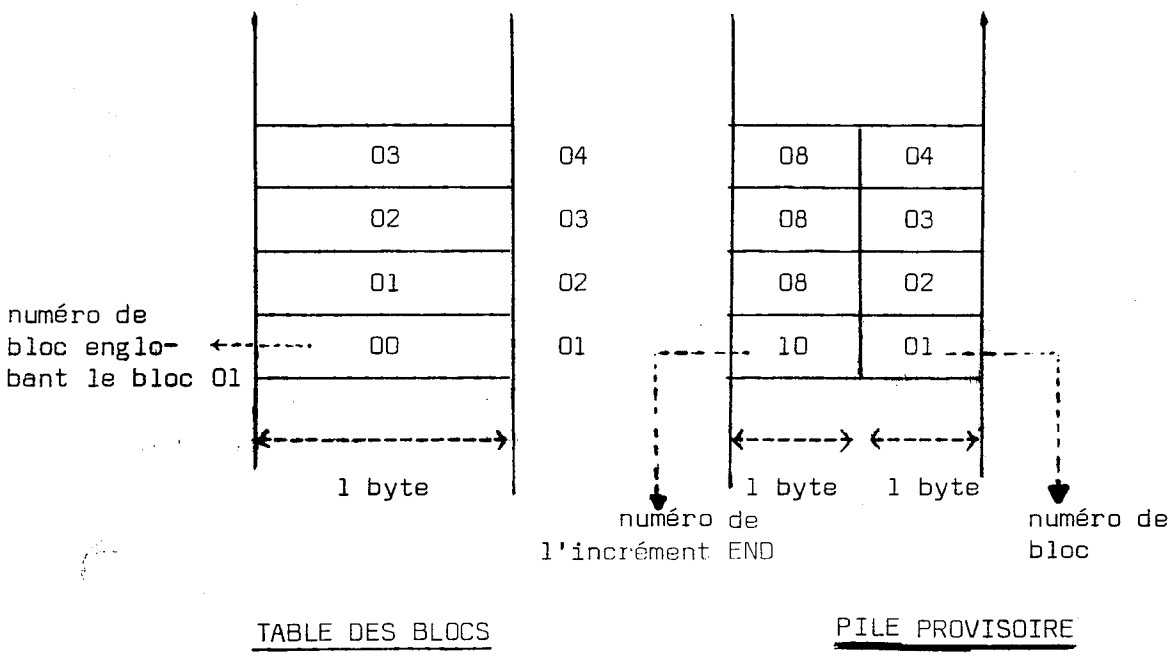
Dès que l'on rencontre un incrément END, on supprime sur la pile toutes les entrées correspondant à ce numéro d'incrément.

La table des blocs est créée à partir de la pile ainsi gérée. Lorsqu'on rencontre un incrément BEGIN ou PROCEDURE, on place dans la table à l'indice correspondant à ce nouveau bloc, le numéro de bloc figurant sur la pile (qui est le numéro de bloc immédiatement englobant). Pour ce nouveau bloc on crée une entrée dans la table.

Considérons cette construction sur un exemple :



Lorsqu'on a examiné l'incrément 06, la table des blocs et la pile contiennent les éléments suivants :



Lorsqu'on a examiné l'incrément 08 END B ;
la table des blocs et la pile sont :

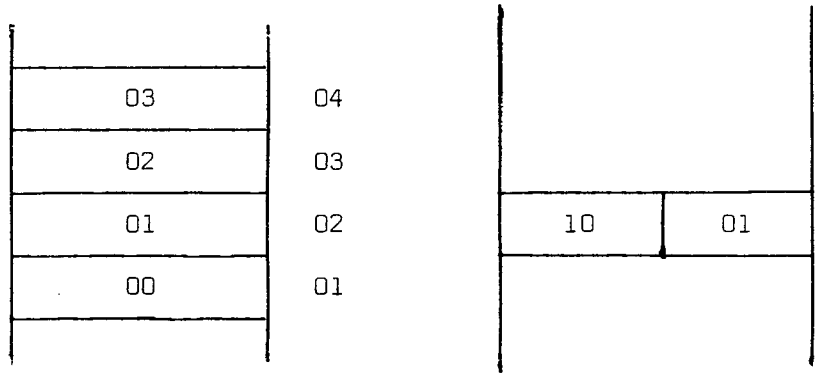


TABLE DES BLOCS PILE PROVISOIRE

On a supprimé sur la pile les trois dernières entrées correspondant aux blocs 2, 3 et 4 qui sont inactifs après l'instruction END B.

La table des blocs est construite au début de la phase de pré-interprétation, et indique que, lorsqu'on traite un incrément du bloc 4, on peut utiliser les déclarations des blocs 3, 2, 1 et 0. En fin de traitement du segment, on conserve cette table de bloc qui nous donnera la chaîne statique des blocs,et le dictionnaire de commande aura la forme :

	Incrément suivant PNTDICT	Type de l'incrément TYPEDICT	Chaînage PREVDICT	numéro de bloc NUMBLOC
01	02	BEGIN	10	01
02	03	.....	00	00
03	04	BEGIN	08	02
04	05	BEGIN	08	03
05	06	.....	00	00
06	07	BEGIN	08	04
07	08	.....	00	00
08	09	END	00	00
09	10	.....	00	00
10	00	END	00	00

Exemple de blocs disjoints

```
01      BEGIN ;.....> bloc 01
02
03      BEGIN ;.....> bloc 02
04
05      END ;
06      BEGIN ;.....> bloc 03
07
08      END ;
09      END .
```

La table des blocs est :

01	03 ←
01	02
00	01

A l'entrée de cette table, correspondant au bloc n° 3, on trouve le numéro de bloc 01. Le bloc 01 est le bloc immédiatement englobant le bloc 03 dans ce programme. Cela signifie qu'à partir du bloc 03, on peut utiliser les informations du bloc 01, et qu'on ne peut pas utiliser celles du bloc 02.

CHAPITRE II

REPRESENTATION DES ELEMENTS DU LANGAGE LORS DE LA PRE-INTERPRETATION

I INTRODUCTION

Le langage PL/1 traite de variables de type arithmétique , de chaînes, de tableaux et de constantes.

Avant de décrire les sous-programmes (modules et routines) permettant de traiter les différentes instructions de PL/1, se pose le problème de la représentation en mémoire des informations utilisées lors des phases de pré-interprétation. Le choix d'une représentation dépend de la manière dont on veut accéder aux informations ainsi que de la nature des modifications à effectuer.

Il faut donc définir les objets élémentaires qui composent les informations et les relations qu'ils ont entre eux (en particulier les fonctions d'accès).

Nous allons, dans une première étape, analyser les informations à utiliser.

Pour ce paragraphe et les suivants, nous utiliserons les définitions de C. PAIR [PAIR] .

Dans la notion de variable il y a deux éléments :

- d'une part : comment la variable est-elle désignée (identificateur simple, identificateur indicé, identificateur qualifié), ceci est la

face syntaxique de la variable

- d'autre part, une variable est un objet d'un certain type. A un tel objet est associée, à chaque instant, une valeur à laquelle la variable donne accès, ceci est la face sémantique de la variable.

Pour des langages à structure de blocs comme PL/1,un même identificateur peut désigner des variables différentes.

Soit, par exemple :

```
Bloc  N° 1 ——→ 01   E  :  BEGIN ;
               02   DCL  A  FIXED  BIN ;
               03   A = 1 ;
               04   BEGIN ;
               05   DCL  A  CHAR (5) ;
               06   X = A ;
               07   END  E ;
```

A l'incrément 03, l'identificateur A désigne une variable arithmétique qui repère des nombres arithmétiques de représentation FIXED BINARY.

A l'incrément 06, l'identificateur A désigne une variable chaîne de caractères qui repère des constantes chaînes de caractères.

A l'exécution on définit les fonctions de repérage qui permettent d'associer une valeur à la variable. Pour préciser davantage à quel niveau s'effectuera la séparation entre les deux phases pré-interprétation et interprétation, examinons le cas des tableaux et celui des structures.

II - TABLEAUX

Un tableau à p dimensions et à N éléments est composé de N variables et d'une fonction qui associe à un p-uple d'entiers une de ces variables.

Exemple -

```
DCL     A (10,5)  FIXED  BIN ;
```

Cette déclaration signifie que pour tout i et j tels que

$$1 \leq i \leq 10$$
$$1 \leq j \leq 5$$

A(i,j) désigne une variable arithmétique de représentation FIXED BINARY.

La représentation d'un tableau à p dimensions est formée :

- de la représentation des variables qui le composent
- de la fonction d'accès qui permet, à partir de tout p-uple d'entiers d'accèder à la représentation du nom associé (c'est-à-dire qui fournit l'adresse de la zone de mémoire représentant le nom associé).

Pour que la fonction d'accès puisse être décrite, il faut que l'on connaisse les bornes du tableau. Ce renseignement,en PL/1, n'est pas toujours connu à la pré-interprétation, mais seulement à l'interprétation puisque, par exemple, la déclaration :

```
DCL   A(N) ;
```

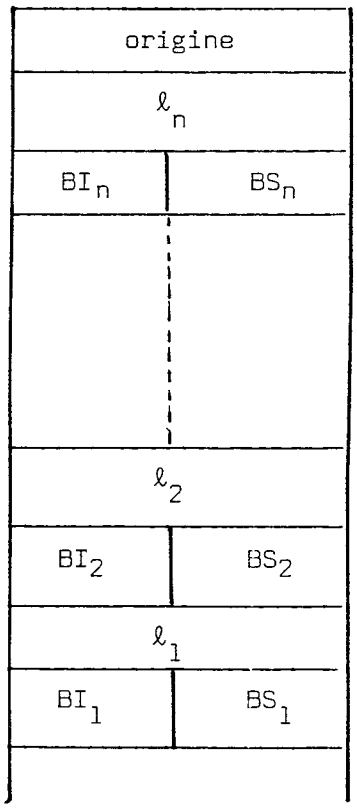
est permise.

Pour construire la fonction d'accès ou fonction d'adressage aux éléments du tableau, nous utiliserons la méthode multiplicative [GRIFFITHS 2]. Les éléments dont nous avons besoin pour construire la fonction d'adressage, en particulier les bornes, ne sont pas connus à la compilation. Les éléments du vecteur

de renseignement (descripteurs du tableau) : bornes inférieures BI_i, bornes supérieures BS_i, la longueur ℓ_i de chaque dimension et l'origine peuvent n'être connus qu'à l'exécution.

L'espace de mémoire nécessaire au vecteur renseignement (partie statique de la représentation du tableau) est déterminé dès l'analyse de la déclaration du tableau, nous le réserverons au moment de la phase de pré-interprétation. Ce vecteur a la forme suivante :

Vecteur renseignement



L'accès aux composantes du tableau (partie dynamique de la représentation) ne peut être défini qu'au moment de l'exécution.

III - STRUCTURES

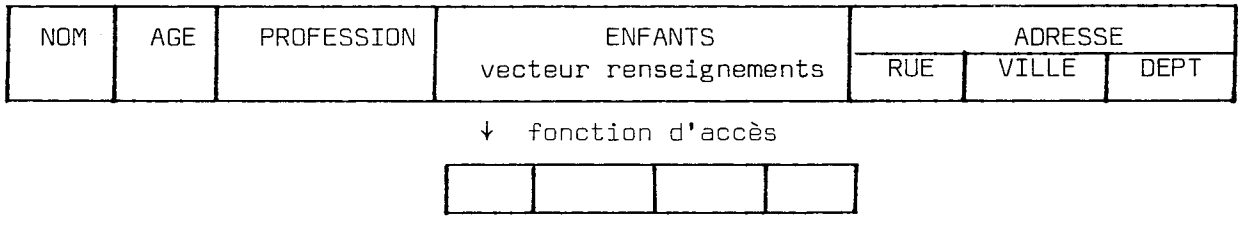
Considérons l'exemple suivant :

```
DCL     1     PERSONNE,
                2 NOM
                2 AGE
                2 PROFESSION
                2 ENFANTS  (N)
                2 ADRESSE
                     3 RUE
                     3 VILLE
                     3 DEPARTEMENT ;
```

Une référence à un nom du niveau 3 serait PERSONNE.ADRESSE.RUE, ou plus simplement, puisqu'il n'y a pas d'ambiguïtés ADRESSE.RUE ou RUE.

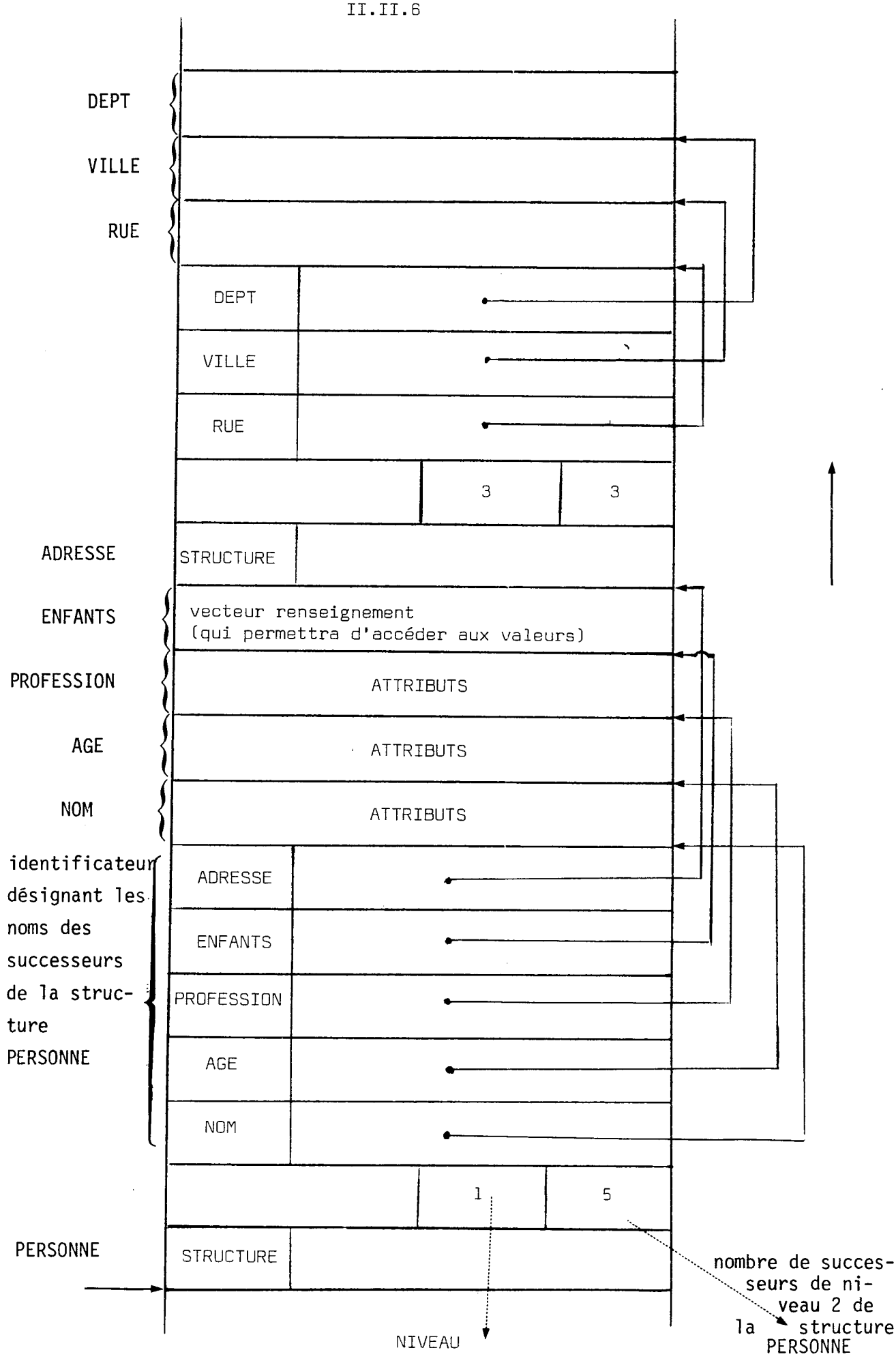
Les éléments statiques de la structure sont représentés de manière contiguë.

Soit schématiquement :



Plus précisément, le domaine de définition de la structure PERSONNE est un ensemble de cinq variables : NOM, AGE, PROFESSION, ENFANTS, ADRESSE.

Comme nous avons une variable tableau dans la structure nous garderons, lors de la phase de pré-interprétation, les éléments représentés en mémoire, de la façon suivante :


DEPT
VILLE
RUE
DEPT
VILLE
RUE
3
3
ADRESSE
STRUCTURE
ENFANTS
vecteur renseignement
(qui permettra d'accéder aux valeurs)
PROFESSION
ATTRIBUTS
AGE
ATTRIBUTS
NOM
ATTRIBUTS
identificateur désignant les noms des successeurs de la structure PERSONNE
ADRESSE
ENFANTS
PROFESSION
AGE
NOM
1
5
PERSONNE
STRUCTURE
NIVEAU
nombre de successeurs de niveau 2 de la structure PERSONNE

Les informations sur les types concernant le champ de la structure PERSONNE, AGE sont accessibles avant l'exécution.

On peut accéder à la description de chaque champ, puisque l'on peut calculer le déplacement par rapport au début de la zone de mémoire où est placée la structure. Toutefois, ce déplacement n'est calculable qu'à la condition que l'on ne place dans cette zone, que les parties statiques (ce qui a été fait dans l'exemple donné pour le tableau ENFANTS à N dimensions).

IV - CONCLUSIONS

Les fonctions de repérage ne peuvent être complètement définies avant l'exécution du programme. Lors de la phase de pré-interprétation il est possible de définir l'accès aux parties statiques des représentations des variables. Dans la phase de pré-interprétation ne sont donnés que les éléments nécessaires pour calculer les fonctions d'accès

(1) vers les parties statiques (descripteur) pour les tableaux

(2) ou vers la représentation des valeurs (dans le cas où tout ce qui est nécessaire est connu avant l'interprétation) : c'est le cas, en particulier, des données de type arithmétique.

Donc, dans cette phase, nous préparons l'interprétation en fournissant, pour chaque variable, l'information utile pour déterminer l'adresse de la zone de mémoire contenant sa valeur.

CHAPITRE III

DICTIONNAIRE D'ATTRIBUTS

I - INTRODUCTION

Le but de ce chapitre est d'examiner les déclarations et de donner les informations contenues dans ce dictionnaire, qui permettent d'effectuer l'association "utilisation-déclaration".

L'organisation du dictionnaire devra être telle que l'on puisse traiter les changements qui sont autorisés au cours de l'interprétation. En effet, le programmeur peut arrêter l'exécution de son programme ; il se place alors dans un état particulier : le "mode" IMMEDIAT. Toute instruction PL/1 (y compris un bloc complet) est alors interprétable. Il est possible d'éditer également un programme en cours d'exécution. Ce qui implique que l'on soit capable d'ajouter de nouvelles déclarations ou de supprimer des déclarations.

Ces différents traitements devront être effectués sans bouleverser complètement l'organisation du dictionnaire d'attributs. Cependant, un changement dans la structure des blocs modifie le programme tout entier et le dictionnaire d'attributs devra être reconstruit.

Les informations seront alors suffisantes pour traiter les classes de mémoire (AUTOMATIC et STATIC) spécifiées dans le langage PL/1.

II - UTILISATION DES INFORMATIONS DU DICTIONNAIRE

Nous avons vu (chapitre I, partie II), qu'une des tâches que nous voulons effectuer dans la phase de pré-interprétation, consiste à établir l'association "utilisation-déclaration". Cette phase permettra d'identifier chaque variable et de préciser, éventuellement, la fonction de conversion à appliquer à sa valeur.

Au préalable, il faut résoudre les deux problèmes suivants : connaître les noms désignés par l'identificateur et connaître les portée de ces noms.

Une déclaration permet d'identifier une variable, elle peut être soit explicite, soit définie par le contexte, soit implicite.

Les déclarations explicites : on aura, par exemple, une déclaration explicite lorsque l'identificateur apparaît dans une instruction DECLARE.

Notons que l'on peut considérer aussi comme déclaration explicite d'identificateur, les trois cas suivants :

a. lorsqu'un identificateur apparaît comme étiquette d'instruction (constante étiquette)

b. lorsqu'un identificateur est dans une liste de paramètres formels d'une instruction PROCEDURE ou ENTRY (identificateur "paramètre").

c. lorsqu'un identificateur apparaît comme étiquette d'une instruction PROCEDURE ou ENTRY (identificateur "nom de point d'entrée").

Les déclarations par le contexte : le langage PL/1 permet de reconnaître les noms désignés par des identificateurs apparaissant dans certains contextes. Par exemple, un identificateur apparaissant dans une instruction GOTO est considéré comme une variable étiquette ; de même, un identificateur dans l'attribut BASED est considéré comme ayant l'attribut POINTER.

Les déclarations implicites : un identificateur est utilisé dans un bloc sans être déclaré explicitement ou par le contexte ; on dit alors qu'il est déclaré implicitement, c'est-à-dire que l'on associe au nom désigné par l'identificateur un type par défaut en fonction de la première lettre de l'identificateur.

Du fait de la structure de bloc du langage, un identificateur est susceptible de représenter, dans un programme donné, plus d'une variable. On appelle portée d'une variable la partie du programme dans laquelle l'identificateur peut désigner cette variable.

Si la déclaration est implicite, la portée de la variable est le segment tout entier.

Si la déclaration est explicite, la portée de la variable désignée par l'identificateur est le plus petit bloc contenant cette déclaration (sont exclus les blocs internes à ce bloc où elle est redéclarée).

Nous avons donc à résoudre les problèmes suivants : association "utilisation-déclaration" d'un identificateur et portée de la déclaration.

Nous avons les deux étapes suivantes à effectuer :

1) Il faut trouver les déclarations explicites (plus précisément les instructions DECLARE) relatives à un bloc.

On fait cette recherche en analysant le dictionnaire de commande, en sautant les incréments relatifs aux blocs internes (ce qui est aisé, en utilisant le chaînage des incréments BEGIN et END, du dictionnaire de commande).

Pour chaque déclaration, une entrée est créée dans le dictionnaire d'attributs, contenant le numéro du bloc de la déclaration.

2) Il faut, à l'aide de l'index (donné par le générateur) associé à un identificateur, trouver quelle est la déclaration correspondante (si elle existe)

valide à cet endroit, c'est-à-dire pouvoir identifier la variable. La détermination de la portée de la déclaration se fera au moyen de la table des symboles.

III - TABLE DES SYMBOLES

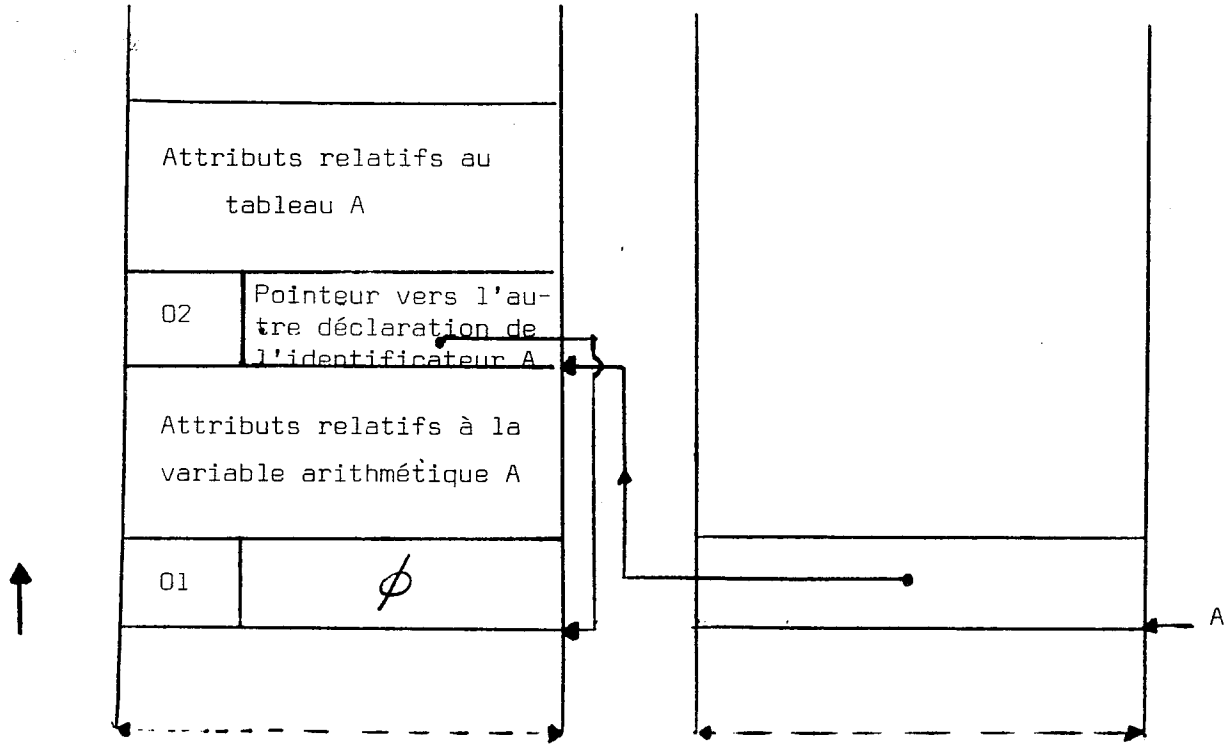
La table des symboles, avec l'aide du dictionnaire d'attributs, nous permet de savoir quels noms sont associés à chaque identificateur (c'est-à-dire à chaque index fourni par le générateur).

La table des symboles comporte une entrée (et une seule), pour chaque index ; elle permet, pour chaque identificateur, d'accéder aux informations du dictionnaire d'attributs relatives à une déclaration de cet identificateur. Pour avoir accès à toutes les déclarations de cet identificateur, nous effectuons, dans le dictionnaire d'attributs, un chaînage entre les différentes déclarations.

Exemple -

```
bloc n° 1 .....  01    BEGIN ;
                 02    DCL  A  FIXED  BIN ; ----> l'identificateur A désigne
                                                 une variable arithmétique
                                                 FIXED BINARY
bloc n° 2 .....  03    BEGIN ;
                 04    DCL  A (10) ; ----> l'identificateur A désigne un
                                            tableau de 10 éléments.
```

L'état du dictionnaire d'attributs et de la table des symboles après le traitement de l'incrément 04 sont les suivants :



A l'adresse de la table des symboles qui correspond à l'identificateur A, on aura un pointeur vers la déclaration de cet identificateur.

A l'aide de la table des symboles, à chaque identificateur, on associe une liste de descripteurs de valeurs pour les variables désignées.

index → VAR1
↓
VAR2
↓
VAR3
↓
etc...

Lorsqu'on a traité toutes les déclarations explicites du segment, on connaît les noms associés à un identificateur.

L'exemple que nous venons de traiter nous montre que l'accès au dictionnaire d'attributs est rapide ; il se fait directement à partir de l'index de l'identificateur.

L'utilisation de la table des symboles s'effectue de la façon suivante.

Soient :

- ISYMBASE le début de la zone réservée à la table des symboles dans la zone de travail de l'utilisateur ;
- INDEX l'index de l'identificateur dont on traite la déclaration.

A l'adresse de la table des symboles correspondant à

ISYMBASE + 4 * INDEX

on aura, soit la valeur zéro, si aucune déclaration de cet identificateur n'a été traitée, soit un pointeur vers la précédente déclaration traitée de cet identificateur.

IV - ELEMENTS DU DICTIONNAIRE D'ATTRIBUTS

4.1 - Définition et portée des variables

Pour résoudre le problème des portées des variables nous enregistrons dans le dictionnaire d'attributs, le numéro de bloc contenant la déclaration.

Nous verrons ultérieurement comment, à l'aide du chaînage des noms et de l'indication du numéro de bloc, il sera possible de déterminer quelle est la variable valide à un instant donné (ch. VIII, partie II).

Il est nécessaire de connaître, pour chaque variable, sa nature (variable simple, indicée, qualifiée ...), son type (arithmétique, etc) et les attributs

relatifs à ce type (FIXED ... etc). Nous voulons que ce processus d'identification de la variable soit fait une fois pour toutes et non pas à chaque entrée dans le bloc. Dans ce but, un dictionnaire d'attributs est défini. Le processus d'identification lors d'une phase de pré-interprétation n'est pas une amélioration spécifique à PL/1 mais peut s'appliquer à tout langage ayant une structure de bloc et compilé de façon interprétative et incrémentale.

4.2 - Types et attributs des données déclarées par une instruction DECLARE

a- Variable simple de type arithmétique

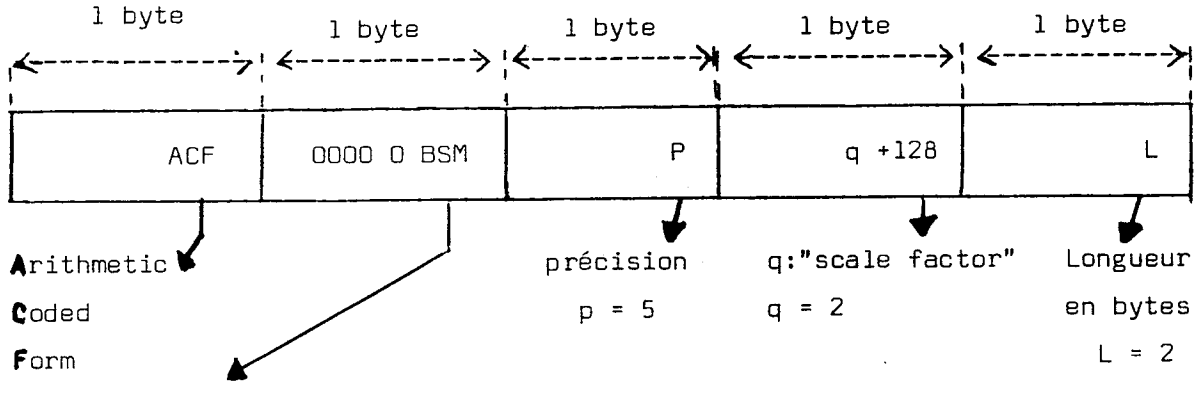
Soit, par exemple, l'instruction :

```
DCL   A   FIXED   BINARY (5,2) ;
```

Rappelons la forme du pseudo-code sous forme de grammaire (en annexe du chapitre I, partie I :

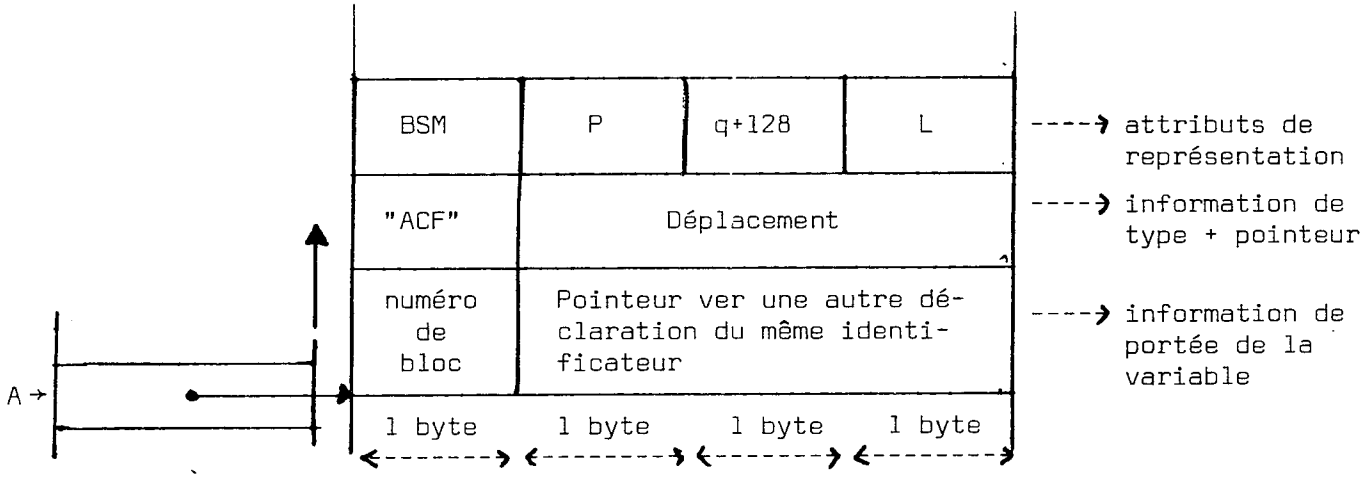
```
   Simpledec  → (Index)  liste-de-type
liste de type → Arithmétique-liste
                Stringlist
arithmétique-liste → (Arithmetic coded form 1) type  (prec scalefactor longueur)
      type      →     base échelle mode
      base      →     (BINARY)
                      (DECIMAL)
      échelle   →     (FIXED)
                      (FLOAT)
      mode      →     (REAL)
                      (COMPLEX)
```

Le pseudo-code donné par le générateur a la forme suivante :




```
B : Base    si  BIN    alors B = 0   / si DEC  alors  B = 1
S : Scale   si  FIXED  alors S = 0   / si FLOAT alors S = 1
M : Mode    si  REAL   alors M = 0   / si COMPLEX alors M = 1
```

Le dictionnaire d'attributs contiendra :



On accèdera au dictionnaire d'attributs grâce à l'index de la variable à l'aide de la table des symboles, qui est construite en même temps que le dictionnaire d'attributs.

b - Nom de tableau

Pour une déclaration de tableau, le pseudo-code contient les informations suivantes :

DIM TYPE [Information spécifique ou type ou attributs] {[expression :] expression [,]}* fin de la liste de dimension.

DIM : nombre de dimensions de tableau

TYPE : type des éléments du tableau

{[expression :] expression [,]}* : bornes inférieures et supérieures du tableau

Le dictionnaire d'attributs ne contient que le type des éléments, le nombre de dimensions du tableau et les informations spécifiques au type. Quant aux informations sur les bornes, nous verrons ultérieurement celles que nous conserverons.

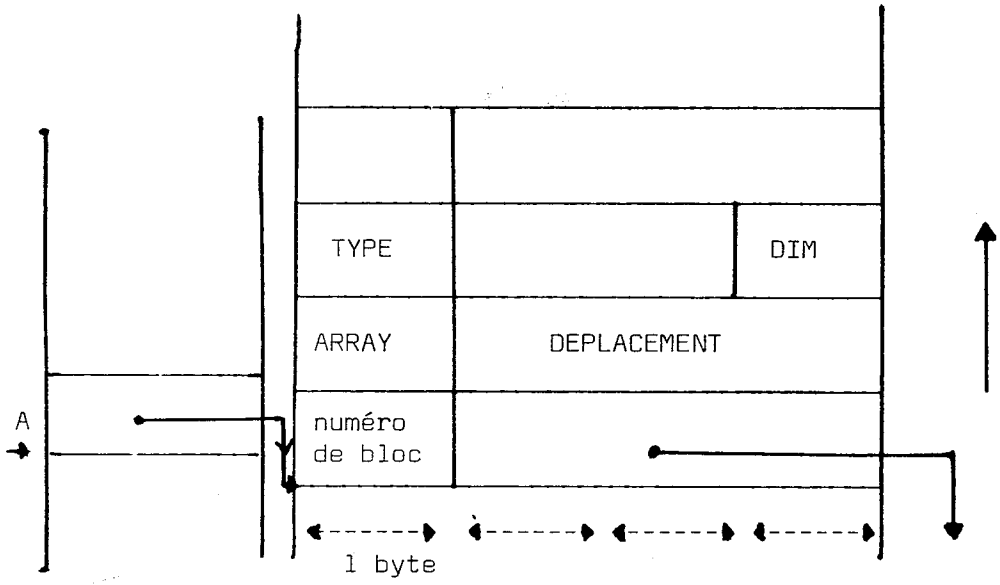


Table des symboles Dictionnaire d'attributs

c - Nom de structure

La grammaire associée au pseudo-code est

Structdec → (Index 4) Elément-de-structure

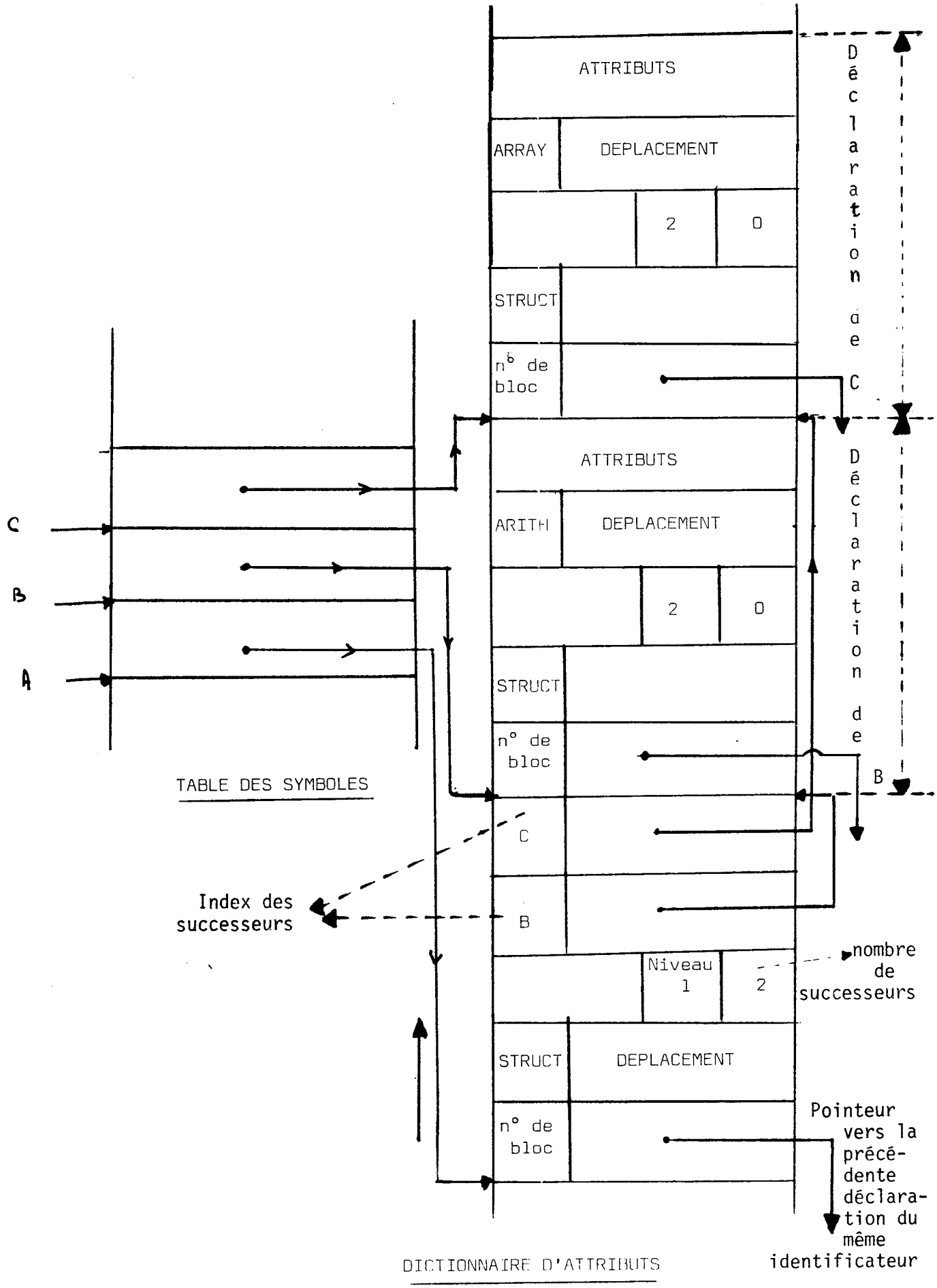
Elément-de-structure → liste-de-structure
liste-de-structure (virgule) élément-de-structure

Liste-de-structure → (Index niveau) liste-de-type
(Index niveau nombre-de-successeur Index*)

Dans le dictionnaire d'attributs nous représentons la structure par un ensemble de pointeurs vers les éléments concernant les identificateurs de la structure.

Exemple - DCL 1 A
2 B FIXED
2 C (N) ;

Dans le dictionnaire d'attributs nous aurons les éléments qui peuvent être parfaitement définis avant l'exécution, soient :


ATTRIBUTS
ARRAY
DEPLACEMENT
2
0
STRUCT
n° de bloc
Déclaration de C
ATTRIBUTS
ARITH
DEPLACEMENT
2
0
STRUCT
n° de bloc
Déclaration de B
C
B
A
TABLE DES SYMBOLES
C
Index des successeurs
B
Niveau 1
2
nombre de successeurs
STRUCT
DEPLACEMENT
n° de bloc
Pointeur vers la précédente déclaration du même identificateur
DICTIONNAIRE D'ATTRIBUTS

Les index des successeurs indiqués dans le dictionnaire d'attributs nous permettent de connaître tous les noms de la structure et de savoir quelle est la déclaration valide pour une variable qualifiée.

Dans l'exemple ci-dessus, la variable qualifiée A.C est une référence au nom C, de type arithmétique, de la structure déclarée ; par contre, la variable A.D n'est pas un élément de cette structure. Le chaînage des noms nous permet de chercher s'il n'existe pas une autre déclaration de structure valide pour cette référence qualifiée.

d - Nom de chaîne

Pour une déclaration de nom de chaîne le pseudo-code a la forme suivante :

Simpledec → (Index) liste-de-type

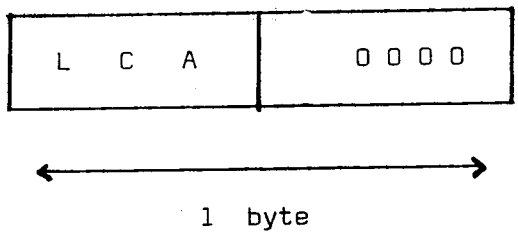
Liste-de-type → arithmétique-liste
stringlist

Stringlist → (4) Type-de-longueur Type-de-chaîne(nombre-d'éléments char*)

Type-de-longueur → (BIT) caractéristique
(CHAR)

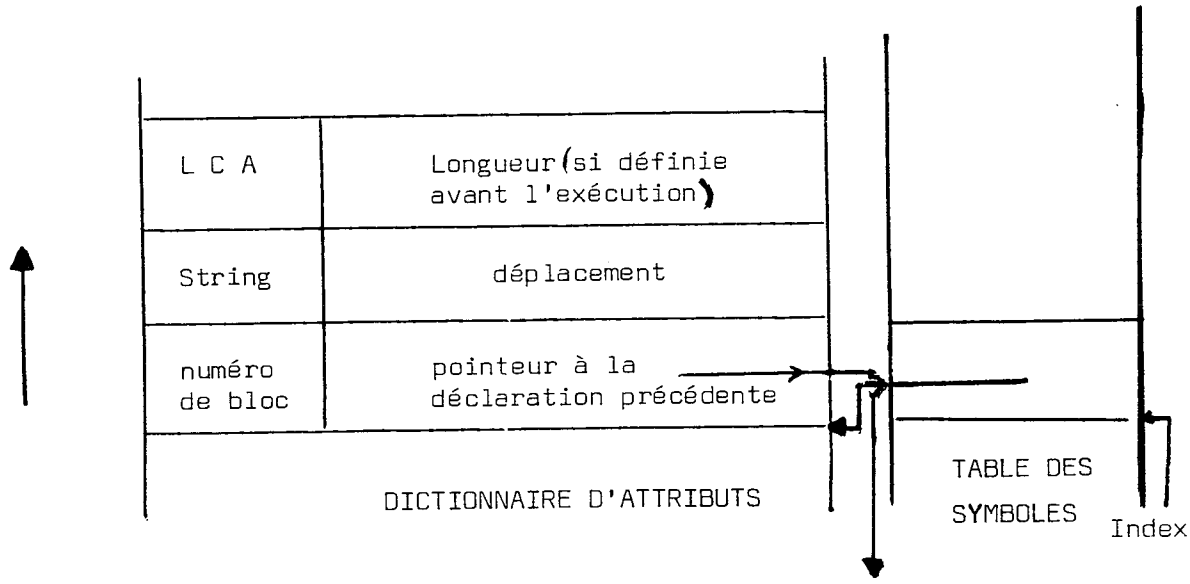
Caractéristique → (UNALIGNED)
(ALIGNED)

ce qui donne comme représentation en mémoire sur 1 byte



L	type de longueur	si attribut VARYING L = 1 / si FIXED 1 = 0
C	type des éléments	si chaîne de bits C = 1 / si caractères C = 0
A	pour les chaînes de bits uniquement	si attributs "unaligned" A = 1 / si aligned A = 0

Le dictionnaire d'attributs a la forme suivante :



4.3 - Déclaration des noms de point d'entrée et des étiquettes

Il est indiqué dans le dictionnaire de commande, si une instruction est étiquetée ou pas.

a - Constante étiquette

Dans le pseudo-code, les étiquettes d'une instruction sont données par la liste des identificateurs qui les constituent, sans indication de type.

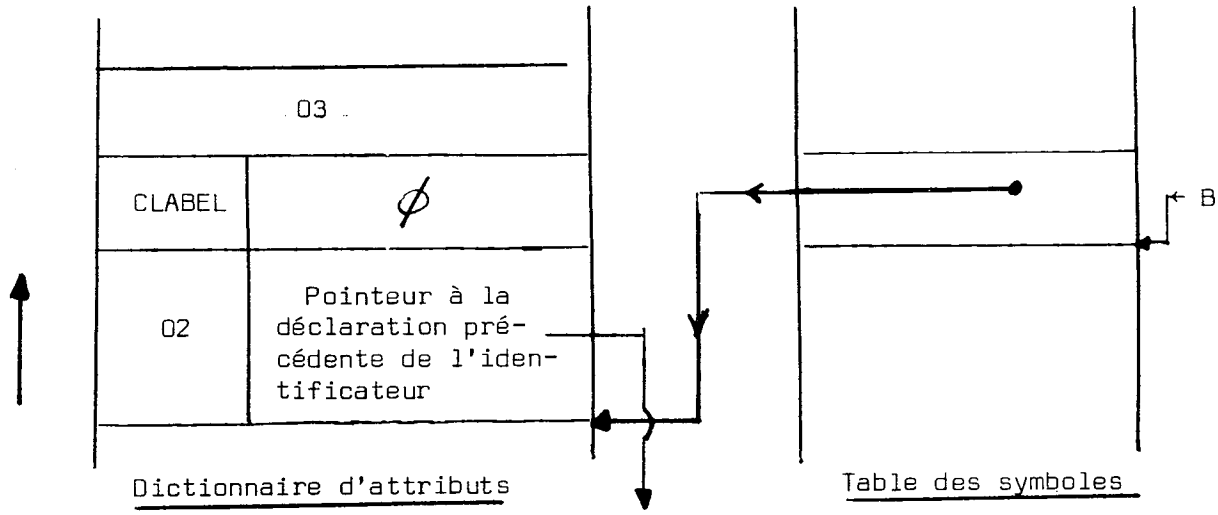
liste d'étiquette → VAR (fin-de-la-partie-étiquette
VAR liste-d'étiquette

Dans le dictionnaire d'attributs est créée une entrée pour chaque identificateur.

Les constantes étiquettes seront indiquées par le type CLABEL et les variables étiquettes déclarées explicitement, par LABEL.

Exemple -

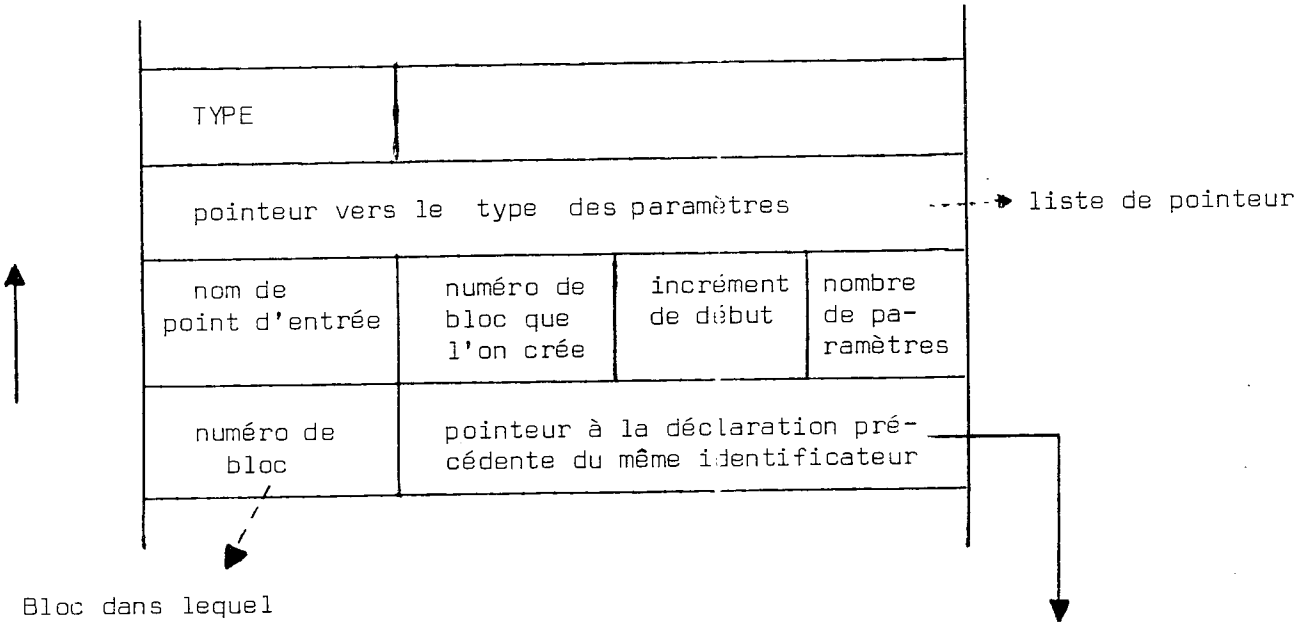
Bloc n°2 02 BEGIN ;
03 B : A = A + 1



La valeur de la constante étiquette sera le numéro de l'incrément qu'elle désigne.

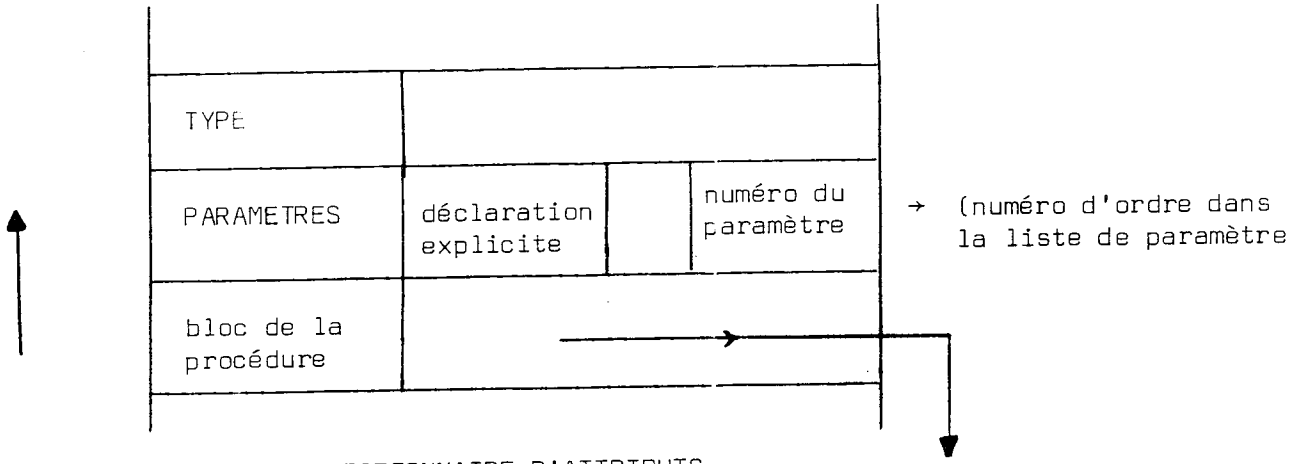
b - Nom de point d'entrée

Pour un nom de point d'entrée, le dictionnaire d'attributs a la forme suivante :



De plus, nous indiquons s'il s'agit d'un nom de point d'entrée primaire ou secondaire.

Le type de chaque paramètre sera indiqué dans le dictionnaire d'attributs.



DICTIONNAIRE D'ATTRIBUTS

L'indication de l'existence d'une déclaration explicite nous permettra d'effectuer les conversions nécessaires, s'il y a lieu, sur les valeurs des arguments à l'exécution.

4.4 - Indications sur les fonctions d'accès

Il importe, dans ce dictionnaire d'attributs, de posséder des informations complémentaires pour définir aisément, lors de l'exécution, la fonction d'accès à la zone de mémoire associée à la variable.

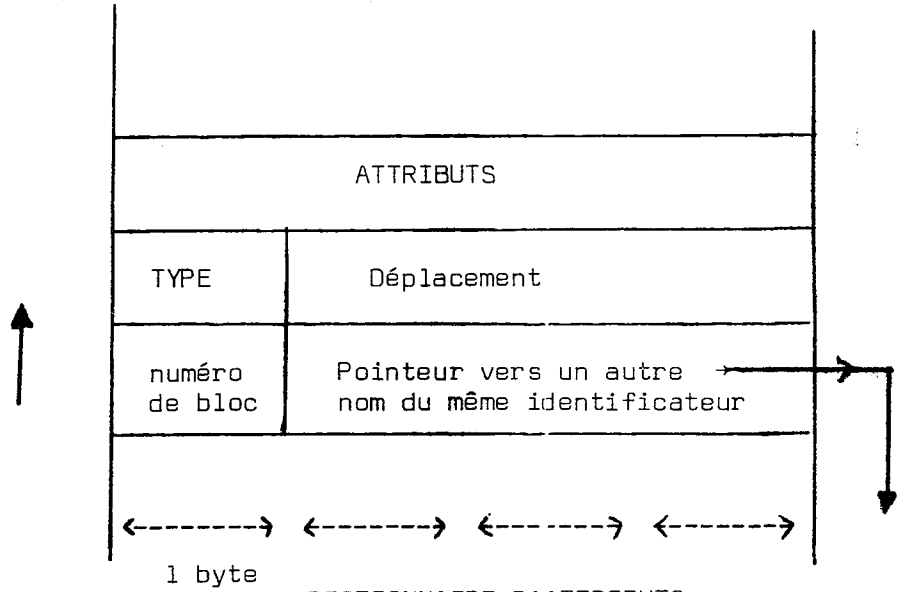
Nous avons vu que, dans le cas d'un tableau par exemple, nous ne pouvions pas définir les fonctions d'accès aux éléments du tableau avant l'interprétation, car les bornes peuvent être des expressions. Le problème est le même pour les chaînes de caractères et les chaînes de bits (car les longueurs peuvent aussi être des expressions).

Mais, par contre, il est possible de définir pour les variables de chaque bloc (au moyen des déclarations explicites), un déplacement par rapport au début de la zone de mémoire utilisée pour un bloc.

Ce déplacement nous permettra d'accéder

- soit à la valeur de la variable, dans le cas où l'on peut calculer, avant l'exécution, la taille de la zone réservée
- soit à la partie statique de la représentation des tableaux (vecteur renseignement), structures ou chaînes.

Ce déplacement sera indiqué dans le deuxième mot réservé à chaque variable. Ce qui donne les éléments suivants,dans le dictionnaire d'attributs.



4.5 - Données complémentaires

L'instruction DECLARE, contient deux types d'informations :

- les informations statiques, par exemple, le type des variables, la partie statique des tableaux et des structures. Ces informations peuvent être traitées avant l'exécution

- les informations dynamiques : il s'agit des informations qui ne peuvent être prises en compte qu'au moment du traitement de la déclaration

  . soit parce qu'elles sont incomplètes (expressions qu'on ne ne peut évaluer),
  . soit parce qu'elles se rapportent à un stade ultérieur de l'exécution (allocation de mémoire).

Les informations dynamiques des déclarations sont indiquées dans le nouveau pseudo-code, et leur existence est signalée dans le dictionnaire d'attributs, de même qu'est donné le pointeur vers leur nouveau pseud--code. Lors du traitement des déclarations nous n'analysons qu'une fois le pseudo-code. Les expressions et les constantes pouvant intervenir dans ces informations sont transformées comme les autres expressions du programme.

4.6 - Cas particulier des variables déclarées STATIC

Les variables déclarées STATIC ne peuvent pas contenir d'expressions (leurs déclarations doivent être parfaitement définies avant l'exécution).

Les déplacements indiqués dans le dictionnaire d'attributs sont donc uniquement des pointeurs vers les valeurs de ces variables.

Ces déplacements sont donnés par rapport au début d'un bloc fictif de numéro zéro qui comprend tout le segment. A chaque instant l'information sur l'espace de mémoire nécessaire à ces variables est conservée en vue de la phase d'interprétation, ce qui permet, en début d'exécution, de réserver la place nécessaire à ces variables.

Si de telles déclarations sont rajoutées en cours d'exécution, nous ne pouvons pas les traiter à moins de recompiler tout le segment.

4.7 - Récapitulation

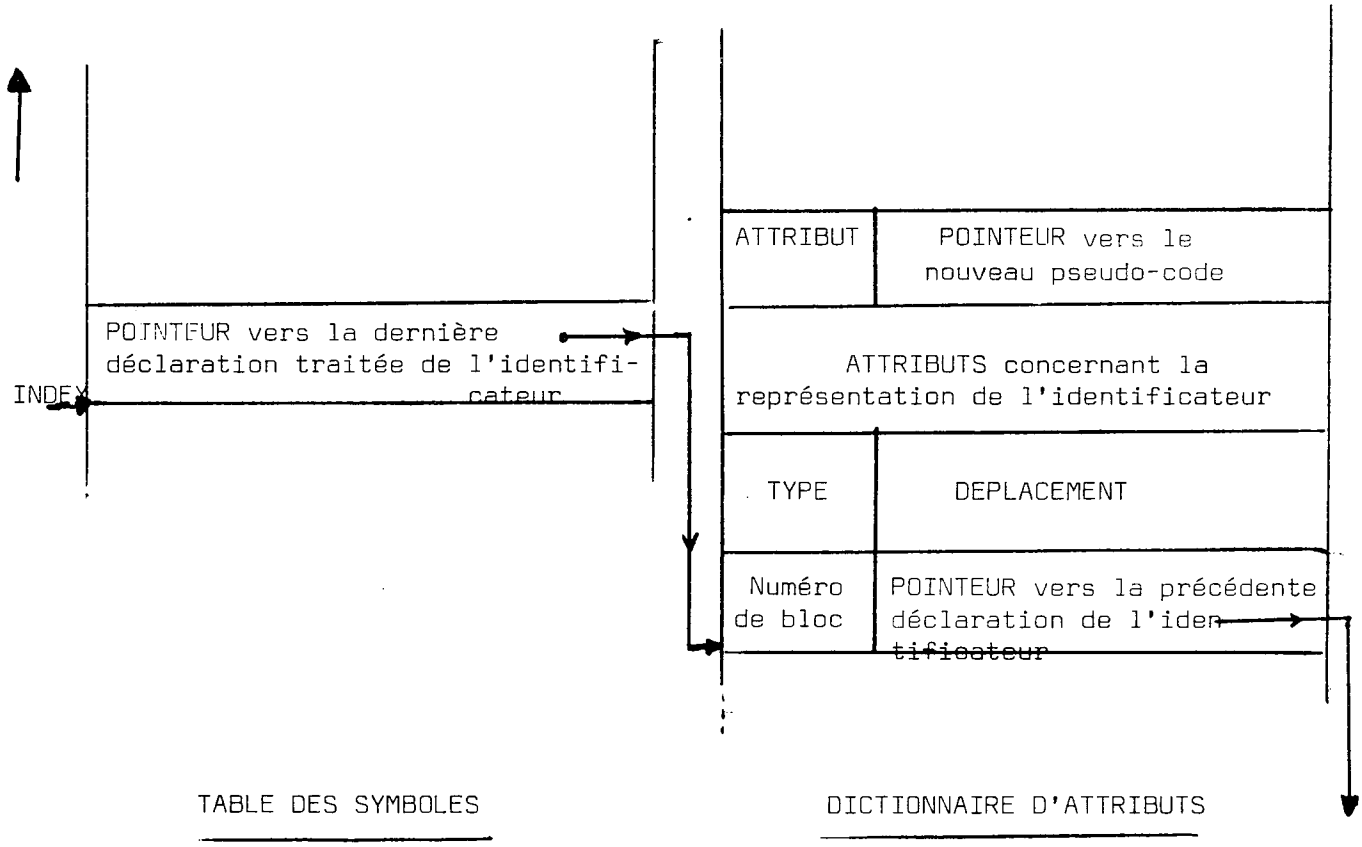
En résumé, pour chaque bloc, nous cherchons les déclarations explicites des identificateurs et nous créons une entrée pour chaque variable déclarée qui comprendra :

- le numéro de bloc de la déclaration
- un pointeur vers la précédente déclaration, si elle existe, d'une autre variable désignée par le même identificateur

- le type de la variable
- le déplacement, dans la pile d'exécution (par rapport au début de la zone de mémoire réservée au bloc), de la valeur ou de la partie statique de la variable.
- éventuellement, un pointeur vers le nouveau pseudo-code, lorsqu'au niveau de la déclaration il existe des attributs (par exemple INITIAL) ou des informations, qui ne sont traités qu'à l'exécution.

En fait, une partie de l'information mise dans le pseudo-code lors de l'analyse de l'instruction DECLARE se trouve conservée dans le dictionnaire d'attributs (telle que type, attributs,... etc) et il devient alors inutile de la conserver dans le pseudo-code.

L'information conservée dans le dictionnaire d'attributs pour chaque variable se présente ainsi :



CHAPITRE IV

LE NOUVEAU PSEUDO-CODE

I - ORGANISATION -

Nous gardons le pseudo-code fourni par le générateur comme image à peu près conforme du programme source, et nous en créons un nouveau qui transmettra à la phase d'interprétation les résultats obtenus par le phase de pré-interprétation.

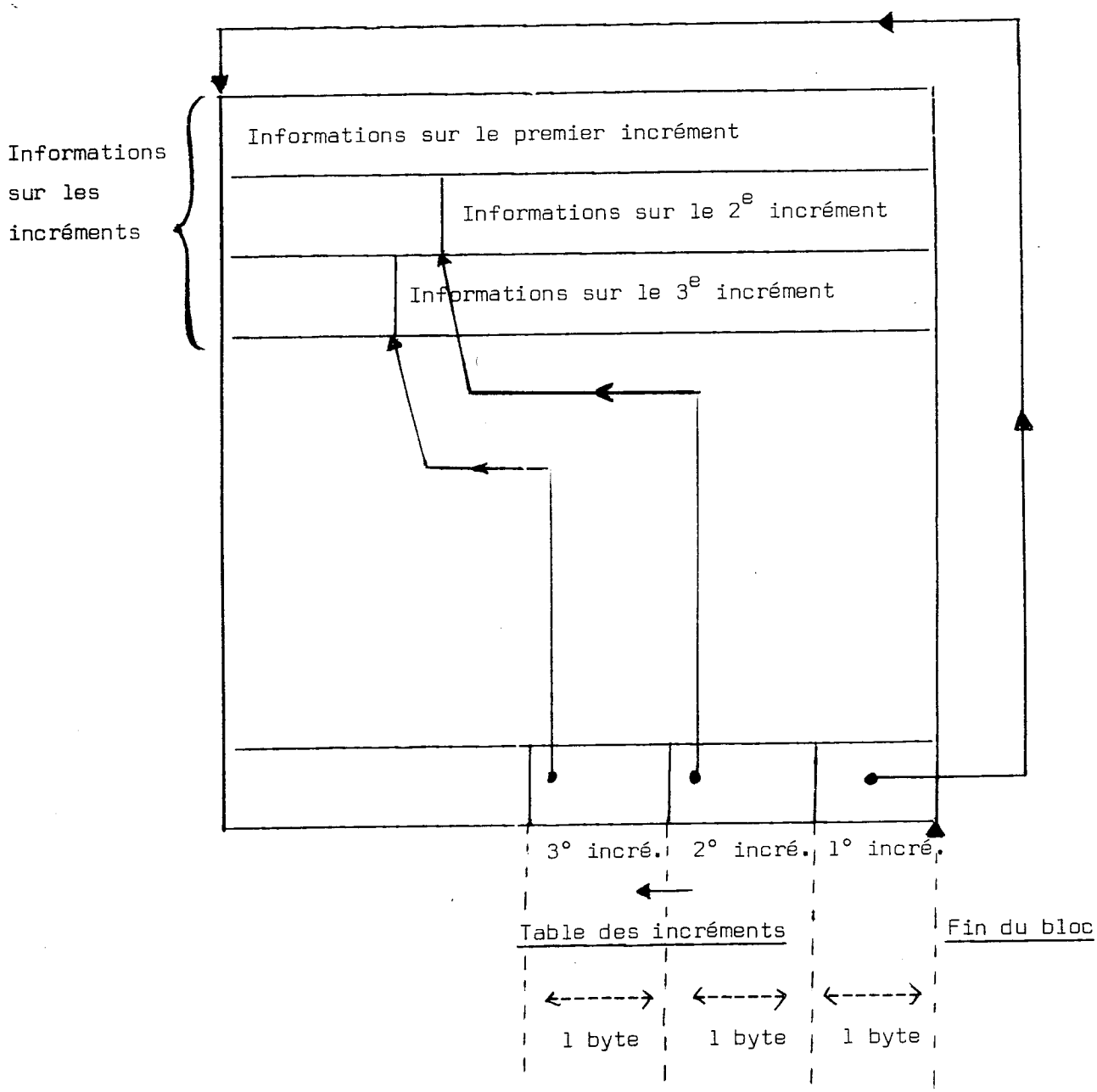
Le nouveau pseudo-code est organisé comme le pseudo-code fourni par le générateur.

Ses éléments sont groupés en blocs de 1024 bytes. Ces blocs peuvent se trouver en mémoire secondaire (disques) ou en mémoire principale.

Dans chaque bloc de pseudo-code il faut accéder à l'information concernant un incrément donné. Dans la phase de pré-interprétation, au fur et à mesure de l'examen des incréments, on crée le nouveau pseudo-code : pour chaque bloc de pseudo-code, on construit une table de pointeurs vers les informations des incréments du bloc. Nous appelerons cette table : table des incréments.

Elle est enregistrée à la fin du bloc et comprend une entrée pour chaque incrément du bloc. Les informations spécifiques à chaque incrément sont enregistrées dans l'ordre de numérotation des incréments.

Ceci donne le schéma d'organisation suivant, pour un bloc de 1024 bytes.



L'adresse de l'information pour un incrément est obtenue de façon simple, à partir du numéro de cet incrément, et du numéro du premier incrément enregistré dans le bloc de pseudo-code. Lorsqu'un incrément n'a pas de pseudo-code, la table ne contient aucune information.

II - PSEUDO-CODE DES DECLARATIONS -

Nous traiterons ultérieurement les modifications apportées par rapport à l'ancien pseudo-code pour les autres incréments. Les modifications apportées au pseudo-code pour les déclarations sont les suivantes :

1. Déclarations explicites.

Lors de la création du dictionnaire d'attributs, pour chaque bloc BEGIN ou PROCEDURE, nous recherchons et traitons les déclarations explicites en analysant le dictionnaire de commande et le pseudo-code fourni par le générateur.

Pour ne pas effectuer une telle analyse à chaque exécution d'une instruction BEGIN ou PROCEDURE, pour chaque déclaration il y a un pointeur vers l'information statique contenue dans le dictionnaire d'attributs (types etc...) concernant cette déclaration, et, éventuellement, si elle existe, l'information dynamique de la déclaration. L'information dynamique est constituée des éléments que l'on ne traitera qu'au moment de l'exécution, soit, par exemple, les expressions intervenant dans l'attribut INITIAL, ou dans les bornes de tableaux. L'information relative aux déclarations se trouve réduite dans le nouveau pseudo-code.

La nouvelle grammaire peut s'écrire :

BEGIN → [12] liste-de-déclarations-explicites

liste-de-déclaration-explicite → déclaration fin-de-déclaration
déclaration liste-de-déclarations-explicites

déclaration → (pas-de-déclaration)
(pointeur)
(pointeur indicateur-d'informations Ex^*)

'pointeur' étant le pointeur vers les éléments de la déclaration enregistrés dans le dictionnaire d'attributs.

Ce qui fait qu'à l'exécution d'une instruction début de bloc (BEGIN ou PROCEDURE) nous avons dans le pseudo-code la liste des déclarations explicites de ce bloc, ce qui permet d'allouer la mémoire nécessaire.

L'ordre dans lequel sont données ces déclarations correspond, lorsque cela est possible, à leur ordre d'exécution. Nous traiterons plus en détail ce problème dans le chapitre suivant.

2. Cas particulier des variables statiques.

A l'interprétation, on alloue en début d'exécution la mémoire aux variables déclarées STATIC.

Nous gardons, comme précédemment, dans le pseudo-code, un pointeur vers la déclaration dans le dictionnaire seulement dans le cas où il y a l'attribut INITIAL. A la première exécution du bloc contenant cette déclaration, l'initialisation est effectuée et nous supprimons l'accès à cette information, c'est-à-dire le pointeur vers les descripteurs de valeurs enregistrées dans le dictionnaire d'attributs.

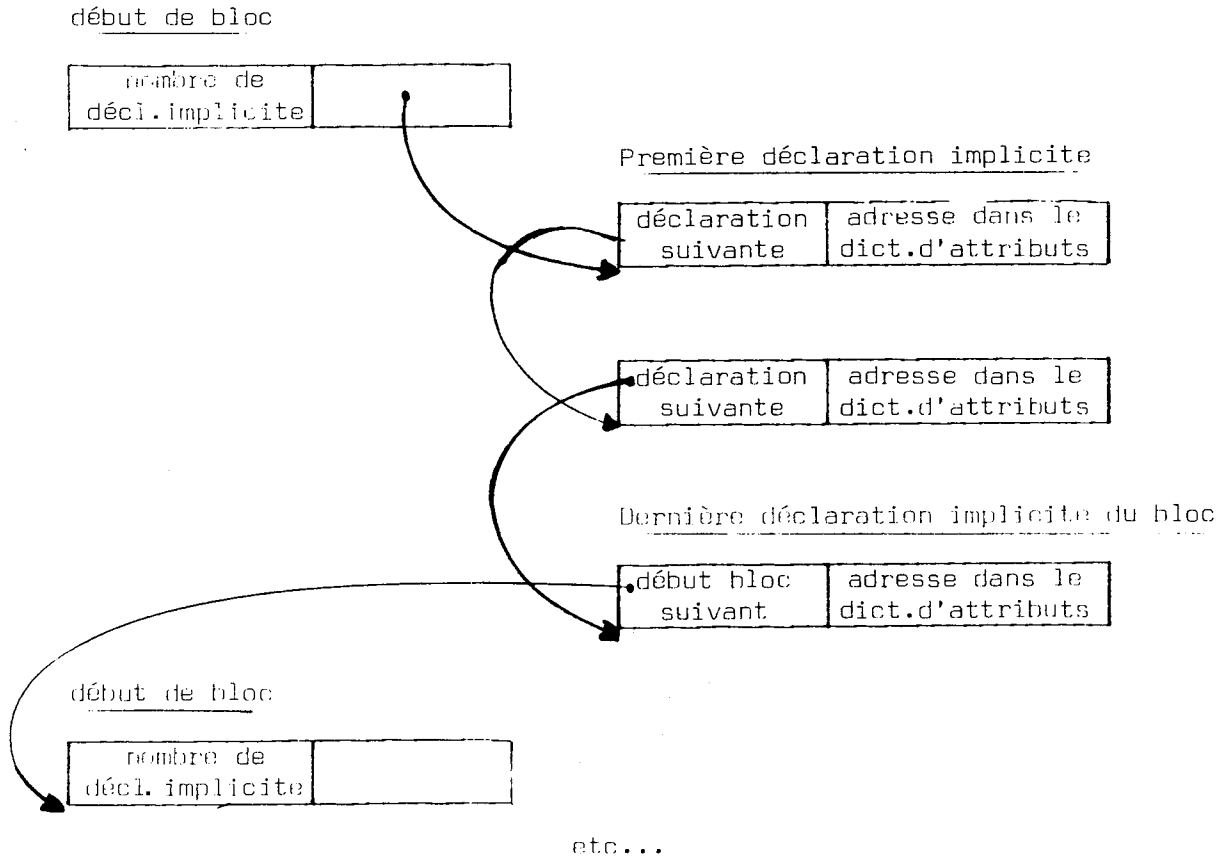
3. Variables déclarées implicitement.

Les identificateurs auxquels ne sont pas associés des déclarations explicites sont considérés comme déclarés implicitement. Les attributs sont mis par défaut en fonction de la première lettre de l'identificateur. La portée de ces variables déclarées implicitement est le bloc de plus externe (comprenant le bloc dans lequel se trouve l'utilisation de cette variable). Nous considérons que le segment tout entier est inclu dans un bloc fictif de numéro zéro.

Nous créons au fur et à mesure que l'on trouve de telles déclarations, une entrée dans le dictionnaire d'attributs. Nous ne pouvons pas procéder comme précédemment en donnant pour l'incrément correspondant à un début de bloc la liste des pointeurs vers les entrées dans le dictionnaire d'attributs correspondant aux déclarations des variables du bloc.

Nous indiquons à ce moment-là un chaînage entre les pointeurs correspondant aux différentes entrées dans le dictionnaire d'attributs. Pour l'incrément correspondant à un début de bloc, nous aurons dans le pseudo-code un demi-mot pour indiquer le nombre de déclarations implicites et, dans le demi-mot suivant, un pointeur vers l'information du pseudo-code concernant la première déclaration implicite, si elle existe. S'il n'y a pas de déclaration implicite, on a un pointeur vers l'information du pseudo-code concernant le début de bloc suivant.

Nous obtenons le chaînage suivant :



BEGIN → [12] liste

liste → (zéro) liste-de-déclarations-explicites
nombre-de-déclarations-implicites impl* liste

impl → pointeur-vers-la-déclaration-implicite-suivante pointeur

Nous indiquons ici la solution que l'on peut adopter bien que nous ne l'ayons pas implémentée.Les déclarations implicites peuvent être traitées dans cette phase de pré-interprétation, c'est-à-dire avant l'exécution.

4. Déclaration par le contexte.

Ces déclarations sont traitées en début de blocs de la même façon que les déclarations explicites. Ce qui est vrai pour le sous ensemble considéré.

CHAPITRE V

REORDONNANCEMENT DES DECLARATIONS

I - INTRODUCTION

Pour effectuer l'identification des variables, on doit examiner toutes les déclarations d'un bloc, et construire le dictionnaire d'attributs. Il est alors possible de réordonner les déclarations, lorsque cela est nécessaire. En effet, des déclarations telles que :

```
        .......
03      DCL     A(M) ;
04      DCL      M    FIXED BINARY INITIAL(5) ;
```

peuvent exister en PL/I, et la déclaration de M ainsi que son initialisation doivent être traitées avant la déclaration de A.

Cette partie n'a pas été implémentée, nous examinons simplement comment ces déclarations pourraient être traitées.

L'exemple ci-dessus montre que dans un même bloc, il peut exister des déclarations interdépendantes. L'ordre des déclarations est alors important et il est nécessaire de le connaître avant l'exécution.

Des déclarations explicites interdépendantes ne peuvent exister que pour des variables automatiques (en effet, les déclarations de variables statiques ne dépendent d'aucun autre identificateur) ayant au moins un des attributs :

- longueur de chaîne,
- bornes de tableaux,
- INITIAL avec facteur d'itération,
- DEFINED,
- LIKE.

Ces attributs peuvent faire référence à d'autres identificateurs. Deux cas peuvent se présenter :

- Ces identificateurs sont déclarés explicitement dans les blocs englobant (et ne sont pas redéclarés dans le même bloc que la déclaration traitée) ; les déclarations n'ont pas à être ordonnées, car ces identificateurs doivent avoir une valeur à l'activation du bloc. De même, si ces identificateurs sont déclarés implicitement, leur portée est alors le bloc le plus externe, soit ici le segment tout entier.

- Ces identificateurs sont déclarés dans le même bloc que l'identificateur dont on traite la déclaration : l'ordre d'évaluation des déclarations, lorsque cela est possible, doit être donné avant l'exécution. Pour l'interpréteur de la première version, M. Berthaud [BERTHAUD] propose de faire, lors de l'analyse des déclarations, une liste des identificateurs intervenant dans ces déclarations. Ces dernières sont examinées cycliquement et l'on pourra traiter immédiatement toute déclaration se référant à un identificateur dont la déclaration a déjà été traitée. Ce travail est effectué chaque fois qu'un bloc est activé.

II - REPRESENTATION DES RELATIONS DE DEPENDANCE SUR UN GRAPHE

Notation :

A → B signifie que dans la déclaration de A, il y a une référence à l'identificateur B, dont la déclaration se trouve dans le même bloc que celle de A. On dira, plus simplement, que "la déclaration de A dépend de celle de B".

Les notations et les définitions sur les graphes sont celles utilisées par C. Berge [BERGE].

1 - Construction du graphe

Considérons les déclarations suivantes :

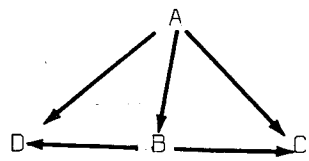
```
03   DCL      A (C)      INITIAL   ((B(D))3) ;
  ...
05   DCL      C      INITIAL   (5) ;
06   DCL      B (D)      INITIAL   ((C)4) ;
  ...
09   DCL      D      INITIAL   (5) ;
```

Soit G (X,U) le graphe orienté associé à ces déclarations pour lequel :

- X est l'ensemble des sommets x_i du graphe. Les sommets du graphe sont les identificateurs dont les déclarations sont interdépendantes. X = {A,B,C,D}.

- U est l'ensemble des arcs du graphe. Il existe un arc entre deux sommets lorsque les déclarations des identificateurs sont interdépendantes.

Le graphe G représentant les relations de dépendance entre les déclarations de l'exemple ci-dessus est le suivant :



Nous considérons la matrice booléenne A associée au graphe G définie par :

- $a_{ij} = 1$ s'il existe un arc allant du sommet x_i au sommet x_j soit $(x_i,x_j) \in U$. x_j est un successeur du sommet x_i.
- $a_{ij} = 0$ s'il n'existe pas d'arcs.

La matrice A est la suivante :

	A	B	C	D
A	0	1	1	1
B	0	0	1	1
C	0	0	0	0
D	0	0	1	0

2 - Propriétés du graphe

a. Propriété 1

Dans la déclaration d'un identificateur A, il n'existe pas de référence à ce même identificateur. Ce qui implique que le graphe G ne possède pas de boucles (circuit de longueur 1), et que la matrice A associée au graphe n'a pas de 1 sur la diagonale.

b. Propriété 2

Le graphe G n'a pas de circuit de longueur 2.

En effet, on trouve, pour la liste d'attributs donnée précédemment, des règles qui l'interdisent. Examinons ces règles pour les différents attributs :

. attribut INITIAL

Il existe une restriction à l'utilisation de cet attribut : si un élément I_1 est utilisé dans une déclaration, dont dépend la déclaration d'un second élément I_2, I_1 ne doit dépendre en aucune manière de I_2 pour sa propre initialisation.

Soit, schématiquement :

$$\text{INITIAL}(I_2 \longrightarrow I_1) \Longrightarrow \neg\ \text{INITIAL}(I_1 \longrightarrow I_2)$$

Exemple :

DECLARE (A(M) INITIAL (1), M INITIAL ((A(1))3) ;
De telles déclarations sont interdites.

. attribut LIKE

La forme d'une déclaration contenant l'attribut LIKE est :
DCL id_1 LIKE id_2 ;

Les propriétés intéressantes pour la relation de dépendance sont :

1. id_2 doit être un nom de structure.
2. id_2 ni aucune de ses sous-structures ne peuvent être déclarées avec l'attribut LIKE. En outre, la déclaration de id_2 (contenant la structure précisée par l'identificateur id_2 ne peut dépendre de la déclaration de id_1 par aucun autre attribut.

$$(id_1 \xrightarrow{\text{LIKE}} id_2) \Longrightarrow \neg\ (id_2 \longrightarrow id_1)$$

. attribut DEFINED

Cet attribut spécifie que des données de type scalaire ou tableau ou structure doivent occuper la même zone de mémoire que celle affectée à d'autres données. Dans de telles déclarations peuvent intervenir des expressions, mais ces expressions ne sont évaluées que lorsque l'on effectue une référence à l'élément défini (c.à.d. à l'identificateur dont la déclaration contient DEFINED).

Les déclarations contenant l'attribut DEFINED n'interviendront pas dans la détermination de l'ordre des déclarations.

. les bornes de tableaux

Soit la déclaration :
DECLARE id $(b_1,b_2,\ldots,b_n)$;

où $b_1,b_2,\ldots,b_n$ sont les bornes du tableau id.

Les bornes peuvent être des expressions qui sont évaluées et converties en quantité entière quand la mémoire est allouée. Ce qui implique que les identificateurs intervenant dans ces expressions, s'ils sont déclarés dans le même bloc que le tableau doivent posséder l'attribut INITIAL.

Schématiquement :

$(id \xrightarrow{\text{bornes}} b_1) \Longrightarrow \exists\, I_1$ dans le même bloc : $(b_1 \xrightarrow{\text{INITIAL}} I_1)$

$(id \xrightarrow{\text{bornes}} b_1) \Longrightarrow \neg\, (b_1 \longrightarrow id)$

. les longueurs de chaîne

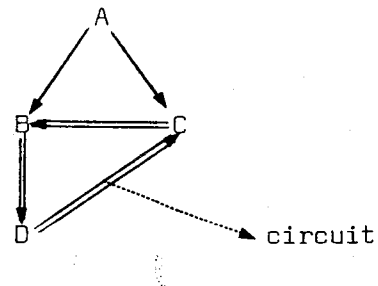
Pour les longueurs de chaîne, on a les mêmes conditions que pour les identificateurs intervenant dans les bornes de tableaux.

c. Propriété 3

On peut généraliser la propriété 2 de la manière suivante :

Le graphe de dépendance des déclarations ne doit comporter aucun circuit. S'il existe un circuit dans le graphe, on ne peut pas établir un ordre entre les déclarations des identificateurs appartenant au circuit.

Soit, par exemple :



Puisqu'il y a existence d'un circuit, nous ne pouvons pas traiter les déclarations de B,C,D, donc les déclarations du graphe {A,B,C,D} et c'est une erreur.

Avant de donner l'algorithme général, nous allons examiner un cas particulier : l'existence de référence à une fonction dans les déclarations.

III - CAS OU LES DECLARATIONS CONTIENNENT PLUSIEURS REFERENCES A UNE MEME FONCTION

Des références à une même fonction apparaissent dans plusieurs déclarations d'un même bloc. L'ordre des appels de la procédure a donc de l'importance, si la valeur retournée dépend de variables globales.

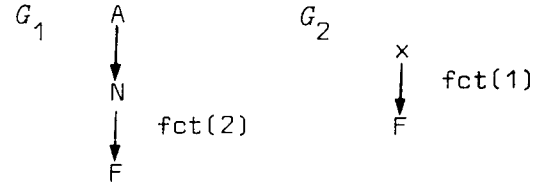
Exemple 1 :

```
01      BEGIN
02      DCL     A(N(1)) ;
                  |----------------------->  Premier appel de F (fonction)
03      DCL     (X(F) ;
                   |---------------------->  Deuxième appel de F (fonction)
04      DCL     N(F)   INITIAL(3) ;
```

Il existe donc une nouvelle contrainte pour réordonner qui est l'ordre d'appel des procédures. Les valeurs de F doivent être connues au moment où l'on traite les déclarations, et la procédure doit être déclarée dans les blocs englobants.

Nous introduisons dans le graphe cette nouvelle relation de dépendance, bien que la déclaration explicite(ou par le contexte) de la fonction ait déjà été effectuée.

Le graphe G associé aux déclarations de l'exemple 1 est :



fct(n) : indique la nouvelle contrainte que l'on envisage pour réordonner. Cette valeur portée sur l'arc signifie que la déclaration d'un identificateur dépend de la valeur d'une fonction, appelée pour la n^{è} fois dans les déclarations du bloc que l'on traite.

Les sous-graphes G_1 et G_2 du graphe contenant tous les deux une référence à une même fonction ne peuvent être traités indépendamment l'un de l'autre.

Le sous graphe G_2 doit être traité avant le sous-graphe G_1.

Ce qui donne comme ordre pour les déclarations dans l'exemple 1 :

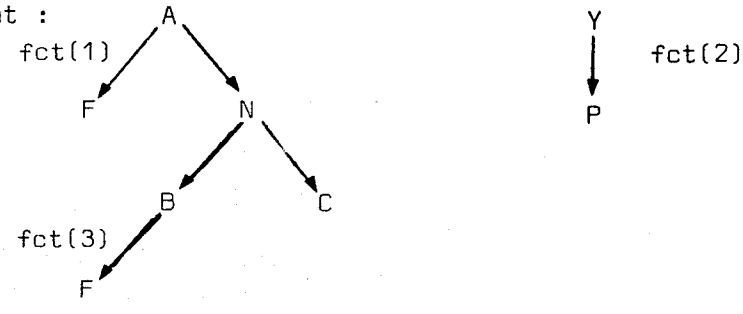
```
                  ┌-----------------------------→ 1er appel de F.
DCL      X(F)
                  ┌-----------------------------→ 2è  appel de F.
DCL      N(F)   INITIAL(3) ;
DCL      A(N(1)) ;
```

Toutefois, il existe des cas où un tel ordre ne peut être trouvé ; donnons en un exemple :

Exemple 2 :

```
                  ┌-----------------------------→ 1er appel de F.
DCL      A(F+N)
DCL      N(B(2)+C)  INITIAL   ((B(2)+C)1) ;
                  ┌-----------------------------→ 2è  appel de F.
DCL      Y(F)
                  ┌-----------------------------→ 3è  appel de F.
DCL      B(F)       INITIAL   ((2)2) ;
DCL      C          INITIAL   (4) ;
```

Le graphe G est le suivant :



Les déclarations de B,C et Y peuvent être traitées avant les autres dans un ordre quelconque, mais B et C doivent obligatoirement être traitées avant N, et ensuite seulement on pourra traiter A. Cet ordre donné ne correspond pas à l'ordre d'appel des fonctions.

```
DCL   C   INITIAL (4) ;

          |----------------------------------------->  2è appel de F.
DCL   Y   (F) ;

          |----------------------------------------->  3è appel de F.
DCL   B   (F)   INITIAL ((2)2) ;

DCL   N   (B(2)+C)   INITIAL ((B(2)+C)1) ;

          |----------------------------------------->  1er appel de F.
DCL   A   (F+N) ;
```

Faut-il alors, lorsque les déclarations sont interdépendantes, considérer pour l'ordre des appels de fonctions, l'ordre donné par le programmeur ?

En réponse à cette question, nous avons essayé de voir ce que donnait le compilateur F. [PL/I].

Exemple 3 :

```
X : PROCEDURE OPTIONS(MAIN) ;
DCL N FIXED(3) ;
DCL F ENTRY RETURNS (FIXED(3)) ;
N = 1 ;
F : PROCEDURE RETURNS(FIXED(3)) ;
N = N + 1 ;
DISPLAY(N) ;
RETURN(N) ;
END ;
DISPLAY(F) ;
BEGIN ;
DCL A(C) INITIAL((B(F))3, (F-2)1) ;
DCL C INITIAL(5) ;
DCL B(5) INITIAL ((F) 1,3,4,2) ;
DISPLAY (' TABLEAU A ') ;
DO I = 1 TO 5 ;
DISPLAY(A(I)) ;
END ;
DISPLAY  (' TABLEAU B ') ;
DO I = 1 TO 5 ;
DISPLAY (B(I)) ;
END ;
DISPLAY (' TABLEAU D ') ;
DO I = 1 TO 5 ;
DISPLAY (D(I)) ;
END ;
END ;
END ;

EXECUTION BEGINS...
     2
     2
     3
     4
     5
     6
TABLEAU  A
3.00000E+00
1.00000E+00
1.00000E+00
1.00000E+00
1.00000E+00
TABLEAU  B
4.00000E+00
4.00000E+00
4.00000E+00
1.00000E+00
1.00000E+00
TABLEAU  D
1.00000E+00
1.00000E+00
1.00000E+00
3.00000E+00
4.00000E+00
```

Conclusions :

L'exemple précédent nous montre que le compilateur F prend comme ordre d'appel des fonctions, l'ordre dans lequel interviennent ces fonctions une fois les déclarations réordonnées.

Nous avons choisi de considérer que l'ordre des appels des fonctions est celui donné par le programmeur.

IV - ALGORITHME PROPOSE

Nous considérons pour l'ensemble des déclarations explicites d'un bloc, le graphe orienté *G* de dépendance des déclarations. Soit A la matrice booléenne associée à ces déclarations.

Ne figurent pas dans cette matrice les identificateurs dépendant d'un autre identificateur par l'intermédiaire de l'attribut LIKE, car ils peuvent être traités dès la pré-interprétation (voir p II.V.19). Si plusieurs déclarations font référence à une même procédure F, on indique l'ordre lexicographique des appels de cette procédure dans un vecteur colonne, F possédant une entrée pour chaque identificateur

Si la matrice A a tous ses éléments nuls, les déclarations explicites du bloc sont indépendantes. Il n'y a pas à réordonner.

4.1 - Recherche de boucles ou circuits

1. Si la matrice A possède des 1 dans la diagonale principale, il y a des boucles sur les sommets (on ne pourra donc pas réordonner). Cela signifie que la déclaration d'un identificateur dépend de l'identificateur lui-même. On cherche les sous-graphes contenant les identificateurs pour lesquels il existe une boucle et on indique au programmeur la liste de ces identificateurs (cf paragraphe 4.2 ci-après).

2. Si la matrice A ne possède pas de 1 dans la diagonale, on recherche tous les circuits du graphe *G*.
   On peut également ne chercher les circuits que lorsque l'on n'a pas pu réordonner les déclarations avec l'algorithme indiqué au paragraphe 4.3 de ce chapitre. A ce moment-là, on cherche les circuits sur la matrice A' obtenue à partir de la matrice A par suppression des lignes et des colonnes correspondant aux identificateurs traités.
   Pour la recherche des circuits, on construit la matrice D, fermeture transitive de A, qui est définie de la façon suivante :

$$D = \bigcup_{k=1}^{n} A^k$$

avec $A^{k+1} = A^k \cap A$

Les éléments dij de D sont tels que :

- dij = 1 si et seulement si aij = 1, c'est-à-dire qu'il existe une suite de sommets $k_1, k_2, \ldots, k_n$, tels que :

$$ai_{k_1} = a_{k_1 k_2} = a_{k_{n-1} k_n} = a_{k_{nj}} = 1$$

- dij = 0 dans les autres cas.

Nous pouvons utiliser par exemple l'algorithme de Warshall pour construire la matrice D [WARSHALL].

Si les éléments de la diagonale principale de la matrice D ne sont pas tous nuls, il existe des circuits. Pour les identificateurs appartenant à un circuit, on cherche les sous-graphes contenant ces identicateurs (cf paragraphe 4.2).

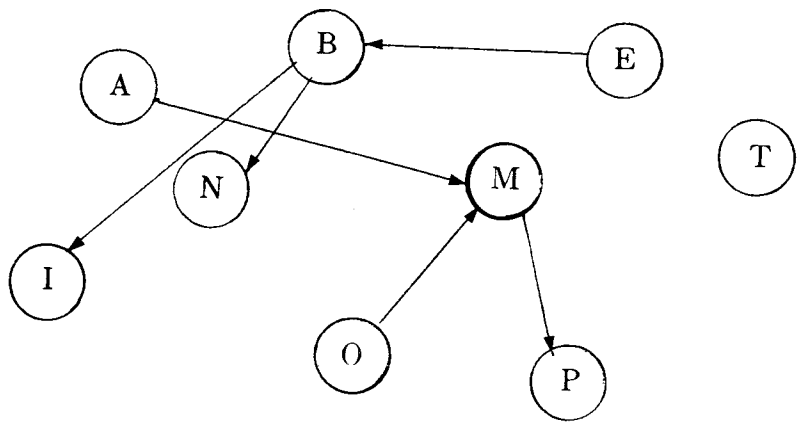
4.2 - Recherche des sous-graphes disjoints

Cet algorithme est donné par Ramamoorthy [RAMAMOORTHY].

Pour appliquer cet algorithme, on transforme chaque sous-graphe en graphe fortement connexe. (un graphe est fortement connexe si pour tout couple de sommet x,y (x,y ∈ X) il existe un chemin allant de x à y et un chemin allant de y à x).

L'algorithme revient à chercher dans le graphe les sous-graphes disjoints fortement connexes.

Soit, par exemple, le graphe



Ce graphe est composé des sous-graphes ayant pour sommets :

S_1 = {A, M, P, O}

S_2 = {B, N, I, E}

S_3 = {T}

4.3 - Recherche de l'ordre

On recherche l'ordre de traitement des déclarations sans tenir compte des déclarations.

On pourrait utiliser l'algorithme du tri topologique de Knuth [KNUTH] pour donner cet ordre. Mais, nous préférons travailler sur la matrice associée au graphe de dépendance, ce qui nous permet de trouver les déclarations qui n'ont pas pu être réordonnées.

Nous nous inspirons d'un algorithme donné par M. Griffiths [GRIFFITHS 3] pour la recherche d'un ordre sur les éléments non terminaux d'une grammaire.

On considère une matrice colonne V ayant une entrée pour chaque identificateur. Les éléments de V sont soit 0, soit 1. L'ordre sera donné dans une matrice colonne R ayant une entrée pour chaque identificateur. On utilise un vecteur U pour marquer les colonnes de la matrice.

On procède de la manière suivante :

1. Initialisation des éléments de V à zéro.
2. n ← 1
3. Recherche d'un identificateur N qui ne dépende d'aucun autre et dont l'entrée dans V soit 0. Lorsque la déclaration de N ne dépend d'aucune autre déclaration, la ligne de la matrice A correspondant à N est nulle. (S'il n'existe pas de ligne nulle, on ne peut pas réordonner (voir remarque 2).
   V(N) ← 1
   R(N) ← n
   On marque la colonne de la matrice A correspondante à N.

4. Tant qu'il existe des lignes nulles pour des identificateurs, dont l'entrée dans V est 0, on reprend à l'étape 3.

5. On supprime les colonnes de la matrice A qui sont marquées.

6. $n \leftarrow n + 1$

7. On répète 3-4-5-6 jusqu'à ce que tous les éléments de V soient nuls.

Remargues :

1. S'il n'y a pas de circuit, il existe obligatoirement une sortie de graphe donc une ligne nulle dans la matrice. Supposons qu'il n'y ait pas de ligne nulle, et considérons un sommet x_1, il existe alors un arc partant de ce sommet allant vers un sommet x_2. Pour x_2, il existe un arc allant de ce sommet soit à x_1, soit à un autre sommet x_3. Comme le nombre de sommets est fini, on a obligatoirement un sommet x_j qui est relié à un sommet x_i du chemin $\{x_1, x_2, \ldots, x_j\}$ donc un circuit.

   Dans le cas où l'on a effectué la fermeture transitive sur la matrice initiale, et qu'il n'y a pas de circuit l'algorithme précédent démarre obligatoirement.

2. Supposons que l'on n'effectue pas la fermeture transitive sur la matrice initiale. S'il n'y a pas de circuit l'algorithme fonctionne normalement. S'il y a un circuit, à l'étape 3 on ne trouvera pas de ligne nulle sur la matrice restante et le vecteur V n'aura pas tous ces éléments à 1. On ne peut pas réordonner.

   On arrête l'algorithme et l'on effectue la fermeture transitive sur la matrice restante pour la recherche des circuits.

4.4 - Traitements des fonctions

S'il n'existe pas dans les déclarations de référence de fonction, c'est-à-dire s'il n'y a pas de vecteur F, on peut donner l'ordre de traitement des déclarations (cf paragraphe 4.5). Dans le cas contraire, on considère la matrice colonne F ayant une entrée pour chaque identificateur. Les éléments de F correspondant au numéro d'appel de la fonction

dans la déclaration de l'identificateur (fct(n) précisé précédemment).

Pour tout vecteur F existant, on effectue l'algorithme suivant :

On considère les vecteurs T et V ayant une entrée pour chaque identificateur.

1. Initialisation des vecteurs T et V à zéro.
   $n \leftarrow 1$ (n étant le niveau des déclarations)
   $n_a \leftarrow 1$ (n_a le numéro d'appel de la fonction).
   $n_t \leftarrow 1$
2. On cherche un identificateur N tel que
   $R(N) = n$ et $V(N) = 0$.

3. Si $F(N) = 0$ On effectue les opérations suivantes :
   $T(N) \leftarrow n$
   $V(N) \leftarrow 1$

4. Si $F(N) = n_a$ On effectue les opérations suivantes :
   $T(N) \leftarrow n_t$
   $n_t \leftarrow n_t + 1$
   $V(N) \leftarrow 1$
   $n_a \leftarrow n_a + 1$

5. Si $F(N) > n_a$ pour tout $R(N) = n$, on ne peut pas réordonner. On donne la liste chaînée des déclarations dans lesquelles interviennent les fonctions ainsi que les numéros de leur appel. On ne pourra traiter ce cas qu'à l'exécution.

6. Tant qu'il existe des identificateurs tels que $R(N) = n$, on reprend à l'étape 2.

7. $n \leftarrow n + 1$
   $n_t \leftarrow n$

8. Tant qu'il existe des éléments de V non égaux à 1, on reprend à l'étape 2.

9. On affecte le vecteur T au vecteur R

R(N) ← T(N) pour tout N.

S'il existe un autre vecteur fonction on reprend l'algorithme avec le nouveau vecteur colonne R. Sinon, on passe au paragraphe 4-5 ci-après.

Exemple :

Si on a obtenu le vecteur R suivant :

$$R = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 2 \\ 1 \\ 2 \\ 1 \end{bmatrix} \qquad F = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 3 \\ 4 \\ 2 \end{bmatrix} \qquad G = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 2 \\ 0 \end{bmatrix}$$

$n = 1 = n_a = n_t$

.R(A) = 1, F(A) = 1 = n_a, T(A) = 1, n_a = 2, V(A) = 1
n_t = 2

.R(C) = 1, F(C) ≠ 2

.R(E) = 1, F(E) = n_a = 2, T(E) = 2, n_a = 3, n_t = 3, V(A) = 1

.R(C) = 1, F(C) = n_a = 3, T(C) = 3, n_a = 4, V(C) = 1
n_t = 4

Ce qui donne pour les vecteurs T et V :

$$T = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \\ 2 \end{bmatrix} \qquad V = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$n = 2$

.R(B) = 2, F(B) = 0, T(B) = 2, V(B) = 1

.R(D) = 2, F(D) = 4 = n_a, T(D) = n_t, V(D) = 1

ce qui donne pour le vecteur R :

$$R = T = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 2 \end{bmatrix}$$

Si l'on applique l'algorithme en utilisant le vecteur G :

.G(A) = G(B) = 0 = G(E)

$$T = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 2 \\ 0 \\ 0 \\ 2 \end{bmatrix}$$

.R(C) = 3, n_t = 3, n_a = 1

G(C) = n_a = 1, T(C) = n_t = 3, $n_{\&}$ = 2, n_t = 4

V(C) = 1

.R(D) = 4, n_t = 4, n_a = 2

G(D) = n_a = 2, T(D) = n_t = 4

$$T = \begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 2 \end{bmatrix}$$

4.5 - Ordre des déclarations

Après construction du vecteur colonne R, on effectue les étapes suivantes :

1. Initialisation :

   n ← 1

   n_t ← 1

2. On cherche l'identificateur N, tel que :
   R(N) = n et l'on effectue les opérations.
   R(N) ← 0
   T(N) ← n_t
   n_t ← n_t + 1

3. Tant qu'il existe des identificateurs tels que :
   R(N) = n on reprend à l'étape 2.

4. n ← n + 1

5. Tant qu'il existe des éléments de R non nuls, on reprend à l'étape 2.

Le vecteur T construit suivant cette procédure, indique l'ordre de traitement des identificateurs.

V - CAS PARTICULIER DE L'ATTRIBUT LIKE

Rappelons la forme d'une déclaration contenant l'attribut LIKE :

DCL id_1 LIKE id_2 ;

où id_2 est un nom de structure, qui peut être qualifié ou non qualifié, mais ne peut pas être indicé.

L'attribut LIKE permet de créer un nouveau nom de structure à partir de la description de la structure nommée id_2. Plus précisément, l'attribut LIKE spécifie que les sous-structures de id_1 ont des composants dont les noms et les attributs sont identiques à ceux de id_2.

Exemple :

```
01   DECLARE      1uA,
                  2B,
                    3Z CHAR(10) ;

02   DECLARE      1u,
                  2w   LIKE A.B ;
```

L'instruction 02 est équivalente à :

```
DECLARE        1u,
               2w,
                 3z  CHAR(10) ;
```

Les numéros de niveau sont modifiés de façon à ce qu'il n'y ait pas de discontinuité dans la description de la structure.

Contrairement aux autres attributs qui interviennent au moment de l'allocation de mémoire aux variables, l'attribut LIKE intervient au moment où l'on réserve les noms. Nous pouvons donc traiter l'attribut LIKE avant la phase d'exécution, et en particulier, au moment où l'on crée le dictionnaire d'attributs.

Traitement de l'attribut LIKE lors de la création du dictionnaire d'attributs.

A l'aide du dictionnaire d'attributs, ainsi que de la table des symboles, nous réservons tous les noms des variables du programme, et il en sera de même pour les nouveaux noms produits par l'attributs LIKE.

Le nom de structure id_2 auquel on se réfère par l'attribut LIKE, peut être déclaré dans le même bloc que le nom de structure id_1, et après celle-ci. Si tel est le cas, le traitement de la déclaration se fera en deux étapes :

Première étape :

On traite la déclaration de la structure jusqu'à l'attribut LIKE, en créant les noms correspondant, ainsi que le résumé des attributs dans le dictionnaire d'attribut. On indiquera l'existence de l'attribut LIKE, pour l'identificateur id_1 on ne pourra donner que le numéro de niveau, mais pas le nombre de ses successeurs. Nous n'indiquerons pas dans le dictionnaire d'attributs le nom qualifié ou pas id_2 mais nous aurons un pointeur vers ce nom dans le pseudo-code. Nous réserverons un mot, dont nous verrons l'utilisation dans la deuxième étape. Nous conserverons d'autre part, la valeur du pointeur au nom id_1, soit $\mathscr{X}$ ce pointeur.

Deuxième étape :

A la fin de l'examen des déclarations du bloc, nous allons chercher quelles sont les sous structures de id_2. Pour cela, nous nous servirons de la valeur du pointeur $\mathscr{X}$, qui nous indique l'emplacement des éléments concernant id_1 dans le dictionnaire d'attributs. Ainsi, nous aurons l'adresse dans le pseudo-code du nom id_2.

Nous construisons pour la structure contenant l'attribut LIKE et pour les sous-structures de id_2 la matrice de dépendance des déclarations et nous vérifions qu'il n'existe pas de boucles (s'il y a des boucles, on ne peut par définir la structure complète).

Dans le cas où il n'y a pas de boucles, nous déterminons alors le nombre de successeur de id_1, et à l'emplacement du pointeur au pseudo-code, nous indiquerons le nom (index) du premier successeur ainsi qu'un pointeur vers ses attributs ; s'il existe plusieurs successeurs, on aura à l'emplacement où aurait dû se situer le nom du deuxième successeur, l'index 00 et un pointeur vers l'endroit où se trouve la liste des successeurs.

Note : L'index 00 indique que l'on doit chercher les noms des successeurs.

Exemple :

```
01   DECLARE    1A,
                  2Z  CHAR(8)
                    3B  LIKE X ;

02   DECLARE    1X,
                  2Y  CHAR(5),
                  2Z  BIT (2) ;
```

Lors de l'examen de la déclaration 01, l'état du dictionnaire d'attributs et de la table des symboles sera celui donné dans la figure I (voir page II.V.23).

Après examen de la déclaration 02, la déclaration complète de la structure est :

```
1A,
  2Z   CHAR(8),
    3B,
      4Y   CHAR(5)
      4Z   BIT (2)
```

Le dictionnaire d'attributs et la table des symboles contient les éléments donnés dans la figure II (page II.V.24).

FIGURE I

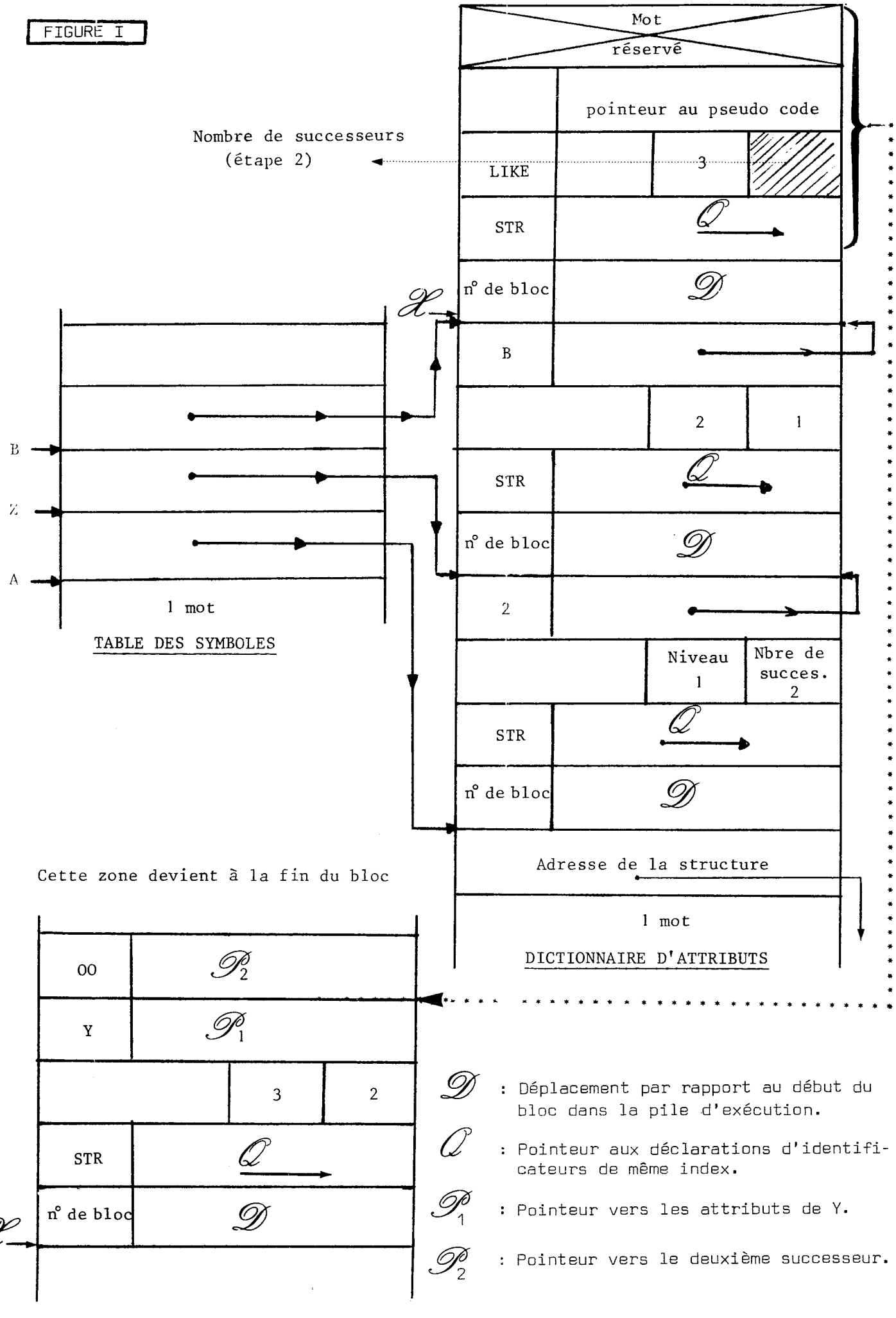
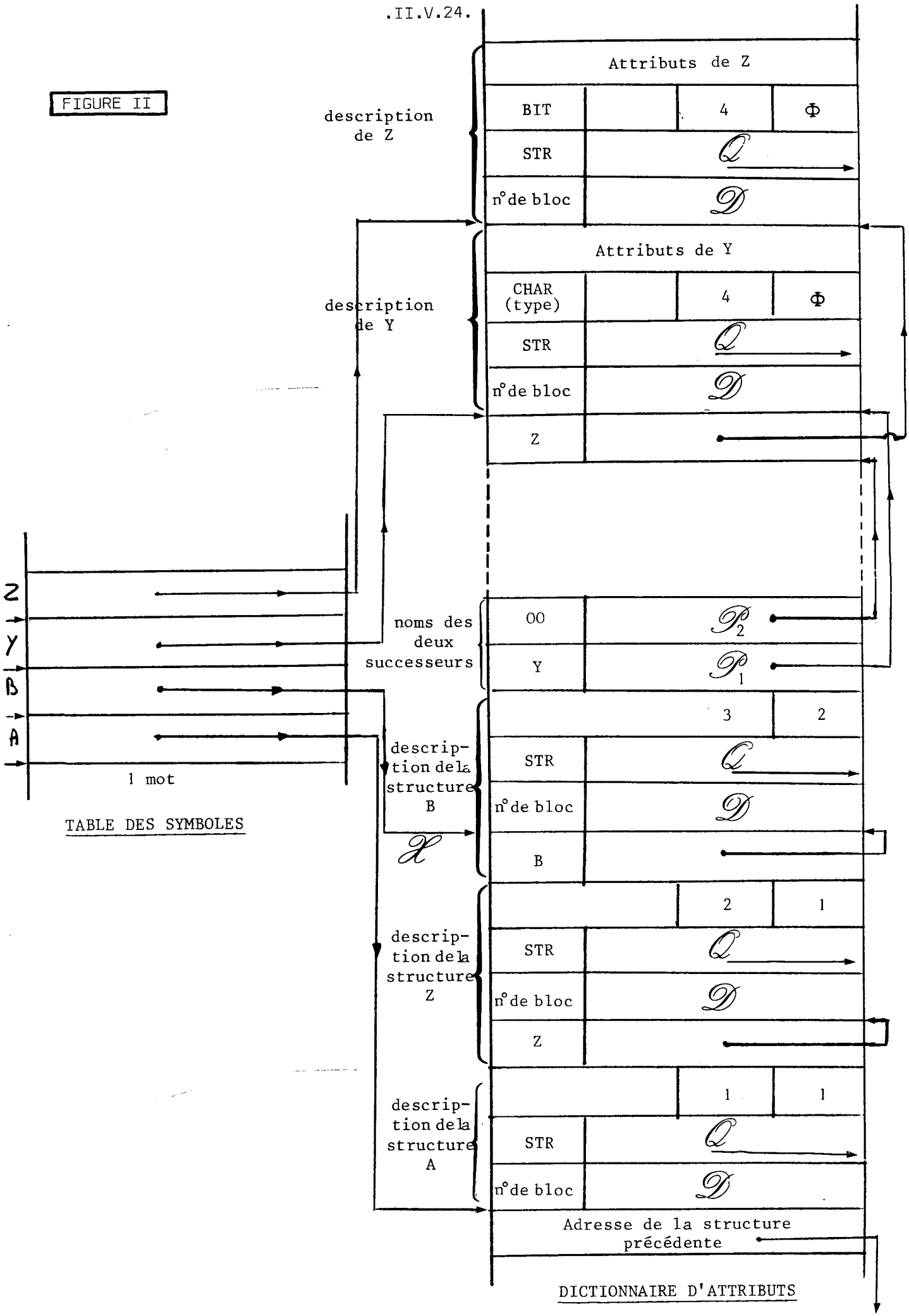

Mot réservé
pointeur au pseudo code
Nombre de successeurs (étape 2)
LIKE
3
STR
n° de bloc
B
2
1
STR
n° de bloc
2
Niveau 1
Nbre de succes. 2
STR
n° de bloc
Adresse de la structure
1 mot
DICTIONNAIRE D'ATTRIBUTS
B
Z
A
1 mot
TABLE DES SYMBOLES
Cette zone devient à la fin du bloc
OO
Y
3
2
STR
n° de bloc
D : Déplacement par rapport au début du bloc dans la pile d'exécution.
Q : Pointeur aux déclarations d'identificateurs de même index.
P1 : Pointeur vers les attributs de Y.
P2 : Pointeur vers le deuxième successeur.

FIGURE II


description de Z
Attributs de Z
BIT
4
Φ
STR
n° de bloc
description de Y
Attributs de Y
CHAR (type)
4
Φ
STR
n° de bloc
Z
Z
Y
B
A
1 mot
noms des deux successeurs
00
Y
3
2
STR
n° de bloc
B
description de la structure B
2
1
STR
n° de bloc
Z
description de la structure Z
1
1
STR
n° de bloc
description de la structure A
Adresse de la structure précédente
TABLE DES SYMBOLES
DICTIONNAIRE D'ATTRIBUTS

CHAPITRE VI

IDENTIFICATIONS ET CONVERSIONS

I - PROCESSUS D'IDENTIFICATION

Le dictionnaire d'attributs a été créé afin de réaliser l'association "utilisation-déclaration" pour les variables (processus d'identification) et de l'indiquer dans le pseudo-code pour la phase de pré-interprétation. En effectuant ce processus d'identification, il est intéressant de tester les conversions à réaliser sur les valeurs et de l'indiquer pour chaque variable.

Pour cela, dans le dictionnaire de commande, il y a lieu d'examiner le type des incréments afin de déterminer ceux dans lesquels peuvent intervenir des expressions.

1.1 - Pseudo-code d'une référence à une variable

Le pseudo-code d'une référence à une variable indique s'il s'agit d'une référence à l'aide d'un identificateur simple, d'un identificateur indicé ou d'une identificateur qualifié.

Ce pseudo-code a la forme suivante :

```
Var       → idsimple   (Index)
            idsubs     (index   nombre-de-dimensions)   Ex*
            Idqual     Id-liste
Idsimple  → (1)
Idqual    → (2)
Idsubs    → (3)
Idliste   →    (nombre-d'identificateurs  index*)
```

1.2 - Recherche de la déclaration valide dans le dictionnaire d'attributs

Le pseudo-code de chaque identificateur donne la nature de chaque référence : identificateur simple, identificateur indicé, etc... Nous pouvons, dès cette phase, déceler un certain nombre d'erreurs dans les références. Par exemple, si l'identificateur est indicé, alors qu'il n'existe pas de déclaration valide de nom de tableau ou de nom de procédure avec paramètre, ou encore si l'identificateur est qualifié alors qu'il n'y a pas de déclaration valide de nom de structure.

L'algorithme qui permet de faire l'association "utilisation-déclaration" pour une variable utilise la table des symboles, la table des blocs et le dictionnaire d'attributs.

Nous désignons par :

- But : le numéro de bloc dans lequel se trouve l'utilisation de l'identificateur.

- Bdec le numéro de bloc (à déterminer) de la déclaration de l'identificateur.

Les différentes phases sont les suivantes :

1) L'index de l'identificateur permet de trouver, avec l'aide de la table des symboles, l'adresse dans le dictionnaire d'attributs de la dernière déclaration traitée relative à cet identificateur.

S'il n'y a pas d'entrée dans la table des symboles pour cet identificateur, il n'existe pas de déclaration explicite de cet identificateur. Il est donc déclaré implicitement. On va à la phase 4.

2) A l'adresse trouvée, le dictionnaire d'attributs fournit le numéro du bloc 'Bdec' dans lequel est effectuée la déclaration. L'un des trois cas suivants peut se présenter.

- Bdec < But

Le numéro de bloc 'dec' de la déclaration trouvée est identique au numéro de bloc 'But' de l'utilisation de l'identificateur. Cette déclaration est la déclaration valide de l'identificateur. On passe alors à la phase 5.

- Bdec > But

La numérotation des blocs est croissante au fur et à mesure de la détection des débuts de bloc (dans la chaîne statique du programme). Le bloc de numéro 'Bdec' est un bloc interne ou disjoint au bloc 'But' de la référence. Et l'on cherche s'il n'existe pas d'autres déclarations de cet identificateur. Cette déclaration n'est pas valide et l'on exécute alors la phase 3.

- Bdec < But

Cette déclaration est valide si 'Bdec' est le numéro d'un bloc englobant le bloc 'But'.

La table des blocs possède une entrée pour chaque numéro de bloc et peut être utilisée pour rechercher le numéro "Ben" du bloc englobant "But" :

a - Si Ben = Bdec, la déclaration trouvée est la déclaration valide de l'identificateur et l'on passe à la phase 5.

b - Si Ben ≠ Bdec, nous cherchons dans la table des blocs le numéro du bloc "Ben1" englobant le bloc "Ben". Deux cas peuvent se présenter :

- le numéro Ben1 est différent de zéro. On reprend alors les opérations a et b en considérant comme numéro "Ben" le numéro "Ben1".

- le numéro "Ben1" est nul, l'on doit passer à la phase 3.

3) Aucun des numéros des blocs englobant le bloc 'But' n'est identique au numéro du bloc de la déclaration trouvée.

Nous allons chercher s'il n'existe pas d'autres déclarations pour cet identificateur. Dans le dictionnaire d'attributs les différentes déclarations d'un même identificateur sont chaînées, ce qui permet d'examiner une autre déclaration : si son adresse est non nulle on reprend l'algorithme à la phase 2 ; si son adresse est nulle, il n'existe pas de déclaration explicite valide pour cette utilisation de l'identificateur ; il est donc déclaré implicitement et l'on passe à la phase 4.

4) Déclaration implicite

Il faut créer une déclaration implicite pour cet identificateur dans le dictionnaire d'attributs. Le numéro de bloc de la déclaration sera zéro. Les attributs par défaut seront appliqués en fonction de la première lettre de l'identificateur (par exemple, si cette lettre est entre "I" et "N" les attributs sont "REAL FIXED BINARY (15,0)", sinon "REAL FLOAT DECIMAL (6)").

5) Déclaration valide

Connaissant la déclaration valide (explicite ou implicite) de l'identificateur, le déplacement par rapport au début du bloc (dans la pile d'exécution) donne l'emplacement mémoire réservé à cet identificateur. De plus, cette adresse (sous forme de déplacement par rapport au début du dictionnaire d'attributs) sera prise comme représentation de l'identificateur dans le nouveau pseudo-code.

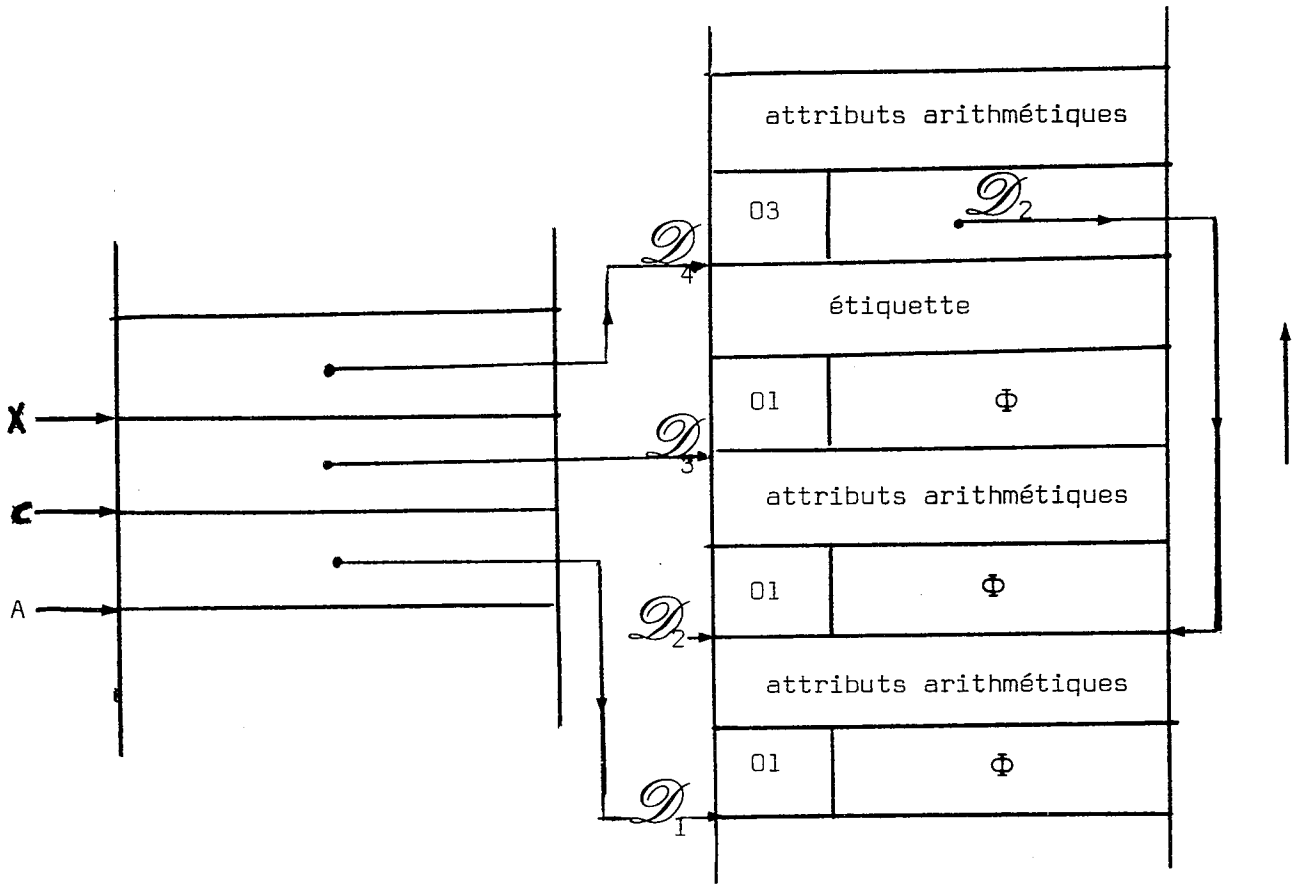
1.3 - Exemple

```
bloc n° 1 --------→  01        BEGIN ;
                     02        DCL   a    FIXED   BIN ;
                     03        DCL   x    FIXED   DEC ;
bloc n° 2 -------→   04        C :   BEGIN ;
                     05        a =   a + x ;
```

```
bloc  n° 3  -------->   06        BEGIN ;
                        07        DCL     x   FLOAT ;
                        08        ....
                        09        END ;
                        10        ....
                        11        END   C ;
                        12        END ;
```

Les seules déclarations de ce programme sont celles figurant dans l'exemple ci-dessus.

Après traitement des déclarations et des blocs dans la phase de pré-interprétation, le dictionnaire d'attributs, la table des symboles et la table des blocs contiennent les éléments suivants :



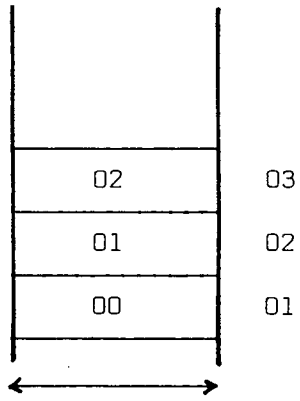


TABLE DES BLOCS

Examinons l'expression se trouvant à l'incrément 05 et recherchons la déclaration valide de l'identificateur A.

Le bloc de l'utilisation de A est le bloc 2 soit But = 2. La table des symboles donne, pour l'index de A, l'adresse $\mathscr{D}_1$ dans le dictionnaire d'attributs (phase 1).

A cette adresse nous trouvons le numéro du bloc de la déclaration B.dec = 01 (phase 2).

B.dec < But (phase 2)

On cherche s'il existe un bloc englobant le bloc n° 2 (B ut) dans la table des blocs.

A l'entrée '02' de la table, on trouve Ben = 01 qui est identique au bloc Bdec = 01 (cas a de la phase 2). On a donc trouvé la déclaration valide de cet identificateur, sa représentation dans le pseudo-code sera $\mathscr{D}_1$.

II - CONVERSIONS

2.1 - Présentation du problème

Considérons l'instruction

```
x = 1 ;
```

dans laquelle x est, par exemple, déclaré 'FIXED BINARY'. Dans la première version de l'interpréteur, l'interprétation d'une telle instruction provoque la conversion de la constante 1 (de type 'FIXED DECIMAL') dans la représentation de x à chaque exécution de cette instruction.

Une telle répétition sera évitée en convertissant la constante dans la phase de pré-interprétation et en donnant, dans le pseudo-code optimisé, la nouvelle représentation de cette constante.

On procédera de cette manière pour toutes les constantes qui interviennent :

- dans les expressions
- comme indices de tableaux et longueurs de chaînes (si nécessaire on réalise la conversion en entiers de ces constantes)
- dans la boucle DO (pas ou valeur finale converties dans la représentation de la variable de contrôle)
- dans l'attribut INITIAL

Dans le cas d'une instruction telle que

```
x = i ;
```

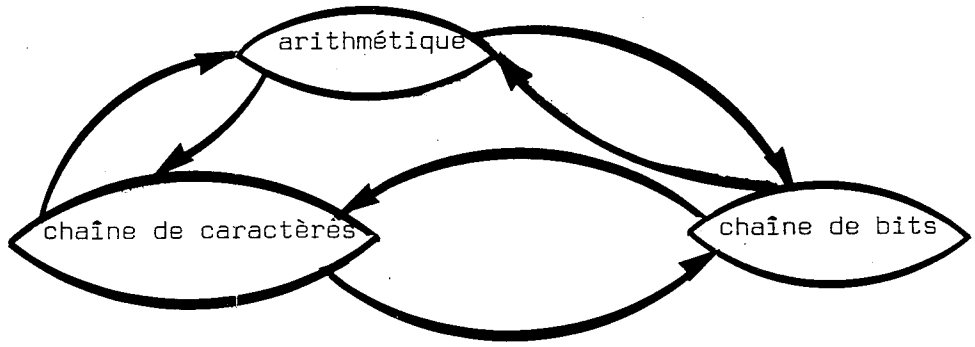
où i est une variable chaîne de caractères et x une variable arithmétique, on réalisera à l'exécution la conversion de la chaîne de caractères en arithmétique pour la valeur de la variable i. La conversion ne peut pas être effectuée dans la phase de pré-interprétation puisque, à cette étape, la valeur de i n'est pas connue, mais sera indiquée dans le pseudo-code.

2.2 - Fonctions de conversion

Trois classes de types primitifs sont considérées pour les conversions : arithmétiques, chaînes de caractères et chaînes de bits.

a - Fonctions de conversion de types primitifs

Des variables de types différents peuvent apparaître dans des expressions. Les conversions éventuelles sont fonction du type des variables et de la nature des opérateurs ; elles peuvent donc être déterminées avant l'exécution. Nous pouvons définir six fonctions entre les types primitifs. Nous les représentons sur le graphe orienté suivant:les sommets sont les types primitifs, les flèches indiquent qu'il existe des conversions entre ces différents types.



b - Fonctions de conversions arithmétiques

Dans la classe de type primitif arithmétique interviennent d'autres conversions. En effet, les variables de type arithmétique se différencient par leurs bases (BINARY / DECIMAL), leurs échelles (FIXED / FLOAT), leurs modes (REAL / COMPLEX) et leurs précisions.

Dans les expressions arithmétiques, certaines conversions sont privilégiées :

- si les bases sont différentes, l'opérande décimal est converti en binaire

- si les échelles sont différentes, l'opérande en virgule fixe est converti en virgule flottante
- si les modes sont différents, l'opérande réel est converti en complexe.

La précision dépend alors de la précision des opérandes et du type des opérateurs. Des précisions différentes pour les opérandes n'entraînent pas de conversions mais interviennent, évidemment, sur la précision du résultat.

Les conversions de base décimale en base binaire, d'échelles FIXED en échelle FLOAT et de mode réel en mode complexe, seront indiquées par les fonctions de conversions (voir ci-après).

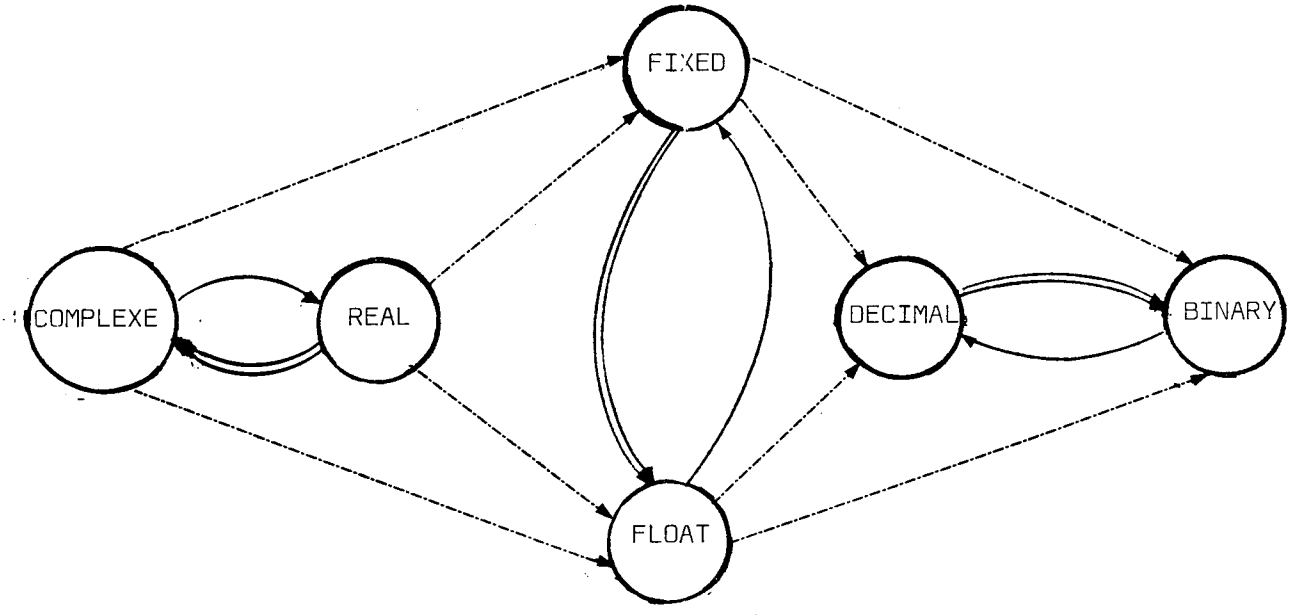
Les conversions inverses de celles que nous venons de citer interviennent dans les instructions d'affectation. Par exemple :

```
DCL   i   FIXED   BIN  ,  A  FLOAT ;

      i = a ;
```

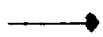
la valeur de a sera convertie d'échelle FLOAT en échelle FIXED.

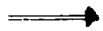
Ces différentes fonctions de conversions peuvent se représenter sur le schéma suivant :



Commentaires :

 représente les liaisons possibles entre les attributs

 représente les conversions à effectuer

 représente les conversions privilégiées dans le cas d'expressions.

Si nous avons deux variables, l'une de type REAL FIXED DECIMAL et l'autre, REAL FLOAT BINARY, le schéma nous indique qu'il y aura deux conversions à effectuer, d'une part d'échelle, d'autre part de base. S'il s'agit d'un opérateur arithmétique, la fonction de conversion sera, FIXED → FLOAT, pour l'opérande d'échelle FIXED, et DECIMAL → BINARY, pour l'opérande décimal.

Dans le cas d'une instruction d'affectation, les fonctions de conversions dépendront de la variable en partie gauche de l'instruction d'affectation.

c - Remarques

Nous n'avons pas indiqué d'erreur sur les types, puisque toutes les conversions entre les différents types primitifs sont permises. Cependant, pour que les conversions puissent être effectuées, il faut que certaines conditions sur les valeurs soient satisfaites. En effet, pour que la conversion d'une chaîne de caractères en chaîne de bits soit possible, il faut que la valeur de la chaîne de caractères ne soit constituée que de 0 ou de 1 ; de même, pour la conversion de chaîne de caractères en arithmétique, il faut que la valeur de la chaîne représente une constante arithmétique ou une expression complexe.

2.3 - Représentation des fonctions de conversions

Dans le pseudo-code optimisé, un identificateur n'est plus désigné par son index, mais par l'adresse dans le dictionnaire d'attributs du descripteur de son nom. Cette adresse est en fait un déplacement (par rapport au début du dictionnaire d'attributs) et occupe trois bytes en mémoires. Nous indiquons la fonction de conversion sur un byte. Cette fonction de conversion peut être con-

sidérée comme un "opérateur unaire" à appliquer à la valeur de la variable. Une fonction de conversion existe aussi pour les résultats intermédiaires et pour le résultat final des expressions. La fonction est indiquée sur l'octet suivant l'opérateur ; il s'agit d'un opérateur unaire à appliquer à la valeur du résultat. L'absence de conversion est indiquée par la valeur nulle. Dans le cas d'une affectation multiple, les fonctions de conversion seront testées à l'exécution.

Envisageons maintenant les formes des différentes fonctions de conversions.

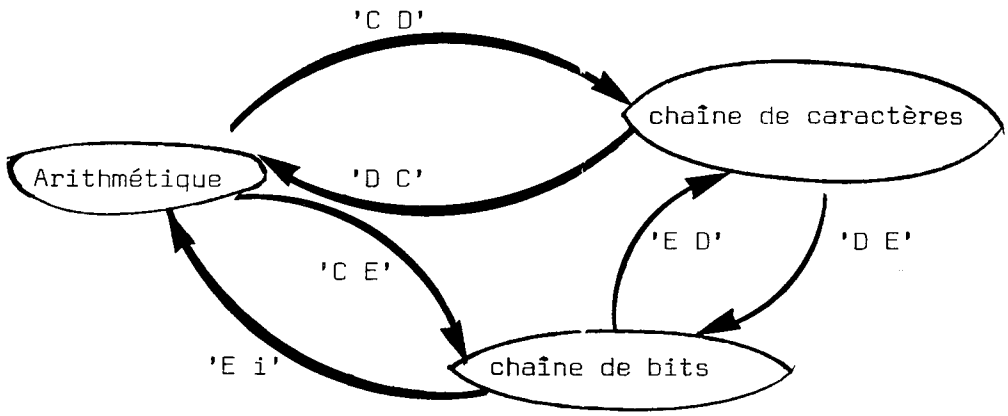
a - Fonctions de conversions entre types primitifs

Les quatre premiers bits (du byte de la fonction de conversion) représentent le type primitif de la variable à convertir et les quatre bits suivants le type dans lequel la variable doit être convertie.

Nous utilisons les valeurs hexadécimales suivantes :

. C pour arithmétique
. E pour chaîne de bits
. D pour chaîne de caractères

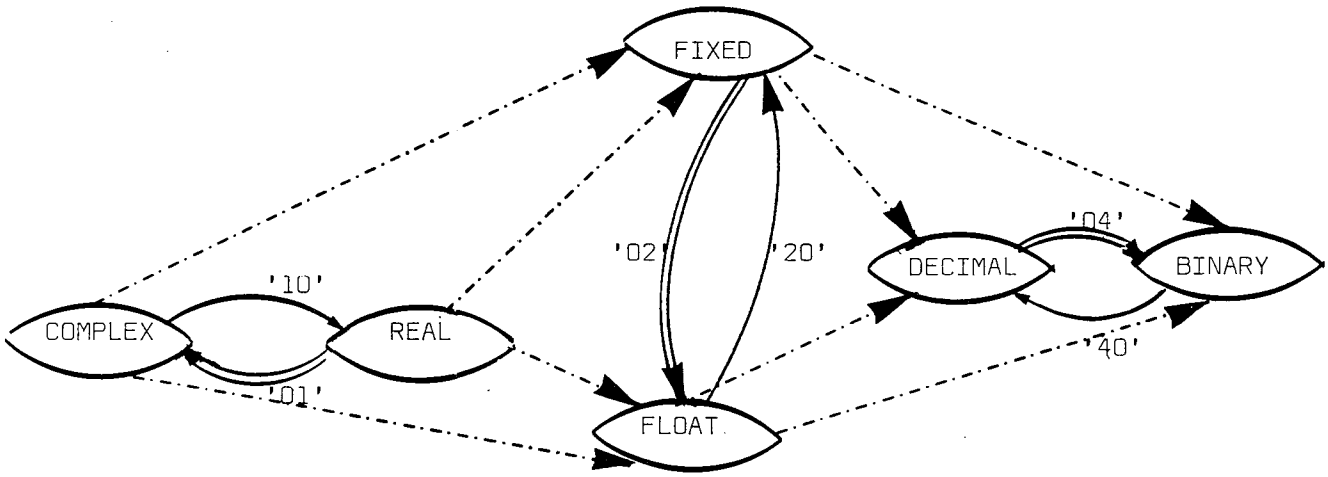
ce qui donne :



La conversion de chaînes de bits en arithmétique ('Ei') peut être précisée de manière plus détaillée : la chaîne de bits est considérée comme une constante FIXED BINARY et les conversions suivantes sont possibles :

- de FIXED BINARY en FLOAT BINARY
- de FIXED BINARY en FIXED DECIMAL
- de FIXED BINARY en FLOAT DECIMAL

b - Fonction de conversion arithmétique



Les fonctions de conversion sont exprimées sur le graphe ci-dessus en hexadécimal (voir en annexe du chapitre les détails des valeurs de ces fonctions).

La fonction de conversion totale à appliquer à la valeur d'une variable s'obtient par l'union de toutes les fonctions de conversion à appliquer à la valeur.

Soit, par exemple, la variable x de type FLOAT DECIMAL dont la valeur doit être convertie en FIXED BINARY. On doit effectuer

- d'une part, la conversion FLOAT → FIXED (20 en hexadécimal)
  ↓
  ou '0010 0000' en binaire

- d'autre part, la conversion DEC → BIN (04 en hexadécimal)
  ↓
  ou '0000 0100' en binaire.

L'ensemble des deux conversions

(FLOAT → FIXED) ∪ (DEC → BIN)

s'exprime '0010 0100' soit 24 en hexadécimal.

III - TRAITEMENT DES EXPRESSIONS

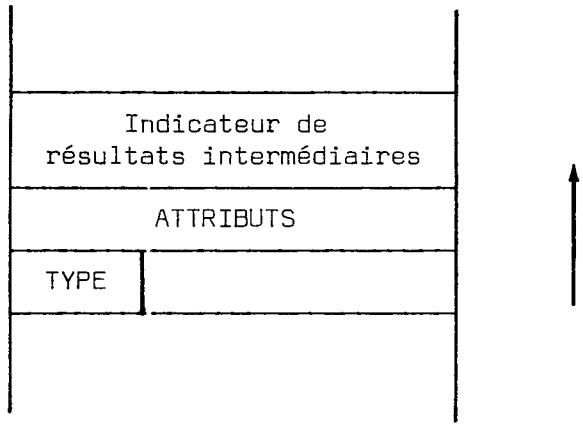
3.1 - Recherche des conversions à effectuer

Le type d'une expression scalaire dépend du type primitif des opérandes et de la classe des opérateurs. Ces opérateurs peuvent être des opérateurs arithmétiques, des opérateurs de comparaison ou des opérateurs spécifiques aux chaînes de bits.

Pour obtenir le pseudo-code optimisé d'une expression, diverses informations doivent être conservées. Elles sont de trois sortes et enregistrées sur une pile :

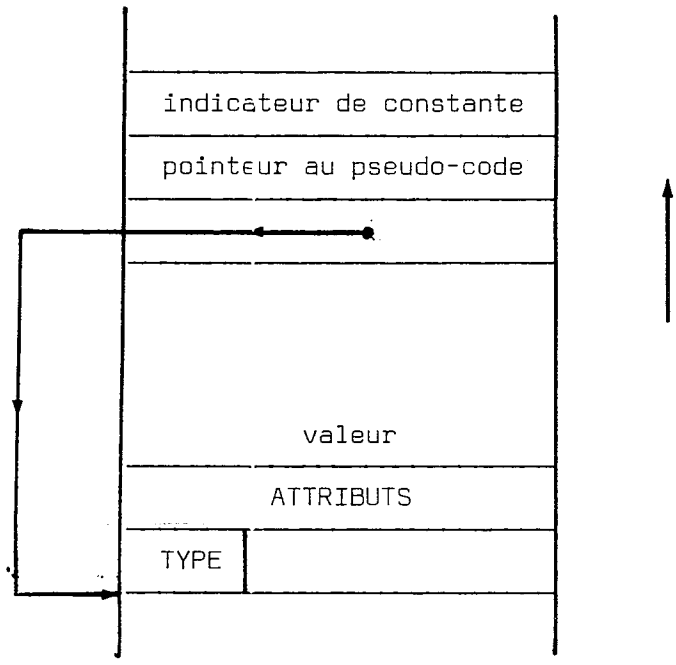
- Si l'opérande est un identificateur, nous enregistrons le déplacement dans le dictionnaire d'attributs vers la déclaration valide de cet identificateur (ce qui donne accès à son type et à ses attributs).

- Si l'opérande est un résultat intermédiaire, nous indiquons sur la pile le type du résultat intermédiaire ainsi que ses attributs. Un indicateur spécial précise qu'il s'agit d'un résultat intermédiaire. Cet indicateur est utilisé pour écrire la fonction de conversion dans le nouveau pseudo-code du résultat intermédiaire (c'est-à-dire au niveau de l'opérateur).



- Si l'opérande est une constante nous indiquons son type et sa valeur. Lorsqu'il y a des conversions à effectuer sur cette constante, la valeur de la constante est convertie et est recopiée dans le nouveau pseudo-code (à l'exécution on n'effectuera plus de conversions sur les constantes, sauf dans le cas d'affectation multiple). Pour cela sont utilisés deux pointeurs : d'une part, un pointeur vers le nouveau pseudo-code, d'autre part, un pointeur vers les caractéristiques de la constante.

Un indicateur spécial précise qu'il s'agit d'une constante, ce qui permet, lorsque des conversions sont à effectuer, de garder la valeur sur la pile, de la convertir et de la recopier dans le nouveau pseudo-code.



En utilisant les informations de cette pile pour les opérandes ainsi que la nature de l'opérateur, nous déterminons la fonction de conversion à appliquer aux opérandes, quatre cas peuvent se présenter :

a) Opérateurs arithmétiques

Ce cas a été précédemment exposé lors de l'énumération des fonctions de conversions.

b) Opérateurs de comparaison

Le langage PL1 établit une hiérarchie entre les différentes catégories de comparaison dans l'ordre suivant :

1. algébrique
2. chaîne de caractères
3. chaîne de bits.

Lorsque les opérandes sont de types différents, l'opérande d'ordre le plus bas est converti au type de l'opérande d'ordre le plus haut.

Pour les comparaisons algébriques les conversions sont effectuées comme pour les opérations arithmétiques, ce qui suppose au préalable les conversions en arithmétique des chaînes de bits et des chaînes de caractères.

c) Opérateur de chaîne de bits

Les opérandes, s'ils ne sont pas du type chaîne de bits, sont convertis en chaîne de bits (lorsque cela est possible).

d) Opérateur de chaîne de caractères

Les opérandes, s'ils ne sont pas du type chaîne de caractères, sont convertis en chaînes de caractères.

3.2 - Exemple de pseudo-code optimisé pour une expression

```
DECLARE        RESULT        BIT (4),
               A   FIXED     DECIMAL (1),
               B   FIXED      BINARY (1),
               C   CHARACTER(2), D BIT(4) ;

RESULT  =  A + B  <  C  &  D ;
```

L'expression est écrite dans le pseudo-code en notation postifixée en tenant compte de la priorité des opérateurs.

```
RESULT  =  AB  +  C  <  D  &
```

Considérons l'évolution pas à pas du pseudo-code optimisé lors de l'analyse de l'expression, en indiquant entre crochets les éléments du pseudo-code relatifs à un élément de l'expression (opérateur ou opérande). Le pseudo-code d'un opérande est :

```
                    (1 byte)                (1 byte)          (3 bytes)
[nature-de-la-référence  fonction-de-conversion  Déplacement dans
 le dictionnaire d'attributs]
```

Le pseudo-code d'un opérateur est :

```
[nature de l'opération (1 byte)  fonction de conversion pour le résultat
 de l'opération (1 byte)]
```

• Soit la sous-expression AB+

Les opérandes diffèrent par leur base ; pour l'opérande A, on a la fonction de conversion 'DEC → BIN' (de base décimale en base binaire).

Le nouveau pseudo-code sera :

[Ref DEC → BIN 'C4' Depl (A)] [Ref '00' Depl (B)] ['+' '00']

Pour l'opérateur '+' nous n'indiquons pour l'instant aucune conversion ('00'), car nous n'avons pas encore testé s'il était nécessaire de convertir le résultat intermédiaire (A+B).

• Soit la sous-expression A+B < C

Le résultat de la sous-expression a la forme arithmétique REAL FIXED BINARY et C est une chaîne de caractères. La comparaison sera donc arithmétique : aucune conversion n'est à effectuer sur A+B et C doit être converti en arithmétique.

ce qui donne :

[Représentation de A et B] ['+' '00'] [Ref CAR → ARITH 'DC' Depl(C)] ['<' '00']

• Soit l'expression finale A+B < C & D

Le résultat de "A+B<C" est une chaîne de bits de longueur 1 , D est une chaîne de bits, l'opérateur est un opérateur de chaîne bits, il n'y a aucune conversion à effectuer ni sur le résultat intermédiaire, ni sur D.

d) Affectation

RESULT = A+B < C & D

Le résultat de l'opération précédente est une chaîne de bits que l'on affecte à une variable de type chaîne de bits. Il n'y a aucune conversion à effectuer sur le résultat final et la fonction de conversion associée à la dernière opération "&" est '00 '. Les conversions intervenant pour la longueur ne sont testées qu'à l'exécution.

ANNEXE AU CHAPITRE VI

- Fonctions de conversion arithmétique

Les conversions privilégiées pour les opérations arithmétiques sont indiquées dans les bits 6,7,8 du byte réservé à la fonction de conversion. Un 1 dans les bits 6, 7 ou 8 indique respectivement, les conversions de base décimale en base binaire, d'échelle FIXED en échelle FLOAT et de mode réel en mode complexe.

La fonction de conversion totale à appliquer à la variable s'obtient par l'union de ces fonctions de conversions. On a alors :

Type de conversion	Fonction (hexadécimal)	Bit 5	Bit 6	Bit 7	Bit 8
modes et échelles différentes (REAL→COMPLEX)∪(FIXED→FLOAT)	'03'	0	0	1	1
bases et modes différents (REAL→COMPLEX)∪(DEC→BIN)	'05'	0	1	0	1
bases et échelles différentes (DEC→BIN)∪(FIXED→FLOAT)	'06'	0	1	1	0
bases, modes échelles différents	'07'	0	1	1	1

- Fonctions de conversions pour les opérateurs de comparaisons

opérande 1 / opérande 2	chaîne de caractères	chaîne de bits	Arithmétique
chaîne de caractères	'0 0' aucune conversion	OPERANDE 1 'ED' conversion de chaîne de bits en chaîne de caractères	OPERANDE 2 'DC' conversion de chaîne de caractères en arithmétique
chaîne de bits	OPERANDE 2 'ED' conversion de chaîne de bits en chaîne de caractères	'00' aucure conversion	OPERANDE 2 'Ei' conversion de chaîne de bits en arithmétique
Arithmétique	OPERANDE 1 'DC' conversion de chaîne de caractères en arithmétique	OPERANDE 1 'Ei' conversion de chaîne de bits en arithmétique	'00' aucune conversion

CHAPITRE VII

PROBLEMES POSES PAR L'INTERACTION

I - INTRODUCTION

Dans le sous ensemble PL/1 que nous avons implémenté nous n'avons pas augmenté les possibilités d'interaction offertes au programmeur. Il est intéressant d'examiner ce que peut apporter à ce niveau l'introduction d'une phase de pré-interprétation.

Les interactions possibles sont de deux types : d'une part, celles qui interviennent avant l'exécution, d'autre part, celles qui interviennent pendant l'exécution.

a) Interaction à l'édition

A l'édition du programme aucune liaison n'est effectuée entre les incréments ; la modification d'un incrément n'entraîne que la modification dans le dictionnaire de commande et dans le pseudo-code des informations relatives à l'incrément modifié. Le générateur n'étant pas modifié il ne se pose aucun nouveau problème pour ce type d'interaction.

b) Interaction à l'exécution

On peut envisager de permettre au programmeur d'exécuter son programme jusqu'à un certain point et d'en arrêter l'exécution.

Il peut alors exécuter immédiatement des incréments ou modifier son programme initial et reprendre l'exécution là où il s'était arrêté.

C'est ce type d'interaction que nous allons examiner dans le cadre de la nouvelle structure du compilateur.

II - INTERACTION LORS DE LA PHASE DE PRE-INTERPRETATION

2.1 - Problèmes posés

La phase de pré-interprétation intervient au début de l'exécution, ou, plus précisément les routines qui la composent sont activées par la commande "EXECUTE ;".

Dans cette phase, un certain nombre d'erreurs globales peuvent être détectées, d'une part sur la structure du segment (blocs non fermés, etc...); d'autre part sur les incréments aux-mêmes (par exemple, instruction GOTO A où A est une étiquette se trouvant dans un groupe DO, etc...).

Le programmeur pourra, en fonction des messages d'erreurs reçus, modifier le programme source ou l'exécuter tel quel.

Examinons les éléments, fournis par la phase de pré-interprétation, qui peuvent être modifiés si on ajoute, enlève ou modifie une instruction.

La phase de pré-interprétation fournit :

- la structure du segment, c'est-à-dire la liaison entre les début et fin de bloc (BEGIN-END, DO-END, etc...)
- Les liaisons entre les variables et leurs déclarations
- la portée des noms créés par les déclarations (dictionnaire d'attributs).

Plusieurs types de modifications, fonction de l'interdépendance des incréments du programme, peuvent être considérés.

a) la modification (insertion ou suppression) d'un incrément qui n'influence pas la signification d'un autre incrément (par exemple, l'instruction d'affectation) n'oblige à recompiler que cet incrément

b) l'insertion ou la suppression d'une déclaration peut changer la signification

des autres incréments se trouvant dans la portée de cette déclaration, c'est-à-dire dans le bloc contenant cette déclaration ; en particulier, les expressions contenant des occurrences des variables dont la déclaration est modifiée, sont à recompiler. On décidera de reprendre toutes les instructions dans la portée de la déclaration modifiée

c) le changement de BEGIN-END de PROCEDURE-END peut affecter la portée des variables du programme et rendre non valide l'association utilisation-déclaration d'une variable faite dans les expressions. De telles modifications conduisent, en général, à recompiler tout le programme

d) le changement des liens IF-ELSE ou DO-END n'affecte, dans le pré-interpréteur, que cette instruction et ne modifie pas les autres éléments fournis par le pré-interpréteur pour les incréments. Le traitement des modifications dans la pré-interprétation peut être effectué à n'importe quel moment du fait de la structure du pré-interpréteur. Afin d'éviter de traiter plusieurs fois un incrément, les modifications seront traitées à la fin de la phase de pré-interprétation.

L'interaction en cours de pré-interprétation est une modification du texte source. La modification est traitée par le générateur qui change en conséquence le dictionnaire de commande et le pseudo-code. Pour connnaître la liste des incréments à traiter nous sommes obligés :

- si l'on supprime un incrément existant ou si on ajoute un incrément, de considérer son type,

- si l'on insère à sa place un nouvel incrément, de comparer son type avec celui du nouvel incrément.

Nous examinons également quels sont les conséquences sur les autres incréments, de la modification faite au programme source.

Exemple :

```
01   BEGIN ;
02   .....
 :
 :
08   END  A ;
```

Supposons que l'on effectue la modification suivante

```
09   SKIP   01 ;
```

Cette commande supprime l'incrément 01 qui est une instruction début de bloc, ce qui modifie la structure du segment.

Mais, par contre, si l'on a aussi les modifications suivantes :

```
05   AFTER   00 ;
05   A  : BEGIN ;
```

On a remplacé un incrément par un incrément de même type. La modification n'affecte que cet incrément.

2.2 - Traitement de l'interaction

Dans une première étape, il faut déterminer quels sont les incréments à traiter ou à reprendre dans la phase de pré-interprétation.

Nous l'indiquerons dans la colonne du dictionnaire de commande (NUMBLOC) introduite dans la phase de pré-interprétation pour la numérotation des blocs. Pour chaque incrément à modifier, le bit 1 (de l'octet réservé à NUMBLOC)sera égal à 1.

Pour effectuer ce travail, il faut que l'incrément modifié soit analysé et que l'on connaisse son type. Le type des incréments se trouve dans le dictionnaire de commande fourni par le générateur ; on accède à cette information

par le numéro de l'incrément. Lorsqu'on supprime un ou plusieurs incréments (commande SKIP) le type de l'incrément supprimé n'est pas perdu, car l'on ne supprime pas l'entrée correspondant au numéro d'incrément ; il suffit de conserver ce numéro. Lorsqu'il s'agit du remplacement d'un incrément par un autre, par l'utilisation successives des commandes SKIP et AFTER, on conserve les mêmes éléments. Avant le traitement par le générateur des incréments modifiés (introduits ou supprimés), les types des incréments du programme initial (c'est-à-dire la copie de la colonne TYPEDICT du dictionnaire de commande), la liste des incréments introduits ou supprimés, les associations BEGIN-END, etc... sont conservés. On garde l'ordre lexicographique initial des incréments (colonne PNTDICT du dictionnaire de commande), pour savoir si un incrément supprimé a été remplacé par un autre incrément.

Il y a un problème supplémentaire lorsque l'incrément supprimé est une déclaration. Pour effectuer les modifications des déclarations dans le dictionnaire d'attributs, il faut connaître l'index de (ou des) l'identificateur. Cette information se trouve dans le pseudo-code fourni par le générateur. Lorsqu'on supprime un incrément, le générateur ne modifie pas l'information dans le pseudo-code, mais en supprime l'accès. Pour conserver l'accès à cette information il faut donc, avant le traitement de la modification par le générateur, conserver, pour la déclaration supprimée, l'accès à ce pseudo-code. Pour la commande SKIP, on est donc obligé d'analyser le nature de l'incrément supprimé, et, si c'est une déclaration, d'enregistrer l'accès à son pseudo-code.

L'algorithme est le suivant :

Phase 1 - Traitements des incréments de type BEGIN-PROCEDURE-END

A partir de la liste des anciens incréments, et de la mise à jour du dictionnaire de commande, nous cherchons,s'il en existe, qui soit de type BEGIN, ou PROCEDURE, ou fin de blocs.

a - Pour de tels incréments, on regarde à l'aide de la copie de l'ordre inidial des incréments (colonne PNTDICT du dictionnaire de commande) s'il remplace un

autre incrément.

a_1 - Si le type du nouvel incrément est identique au type de l'ancien incrément, on vérifie si la structure du segment n'est pas modifiée. On regarde si les associations BEGIN-END, PROC-END, DO-END, ne sont pas changées. Cela peut se produire dans le cas d'instruction BEGIN que l'on étiquette, ou si l'on associe à l'instruction END une étiquette différente (END A remplacée par END B. On refait l'association BEGIN-END, si les liens sont modifiés, et l'on reprend l'interprétation pour tout le programme.

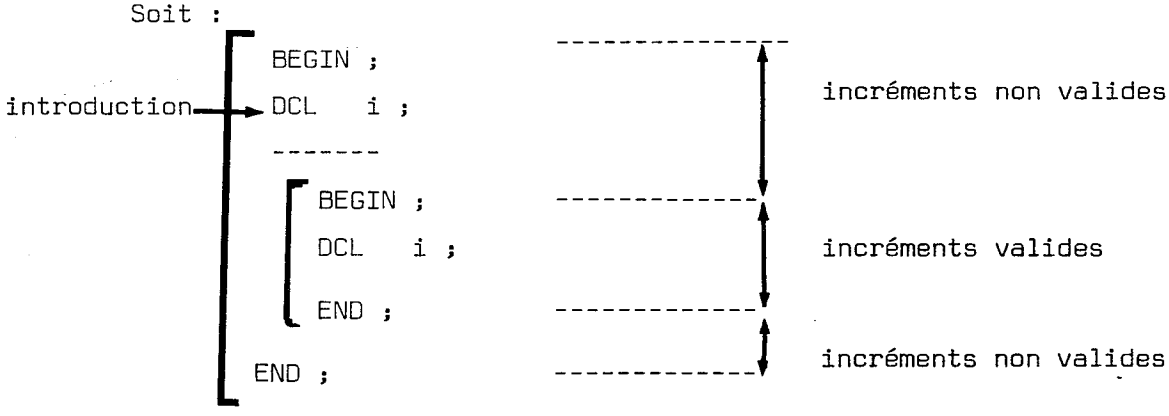
a_2 - Si les types sont différents, la structure du programme est modifiée et l'on reprend la pré-interprétation pour tout le programme.

b - S'il s'agit d'introduction ou de suppression d'incréments de ce type, la structure du programme est modifiée, et l'on refait la pré-interprétation pour tout le programme.

Phase 2 - Traitement des déclarations

Lorsqu'un incrément de type déclaration est supprimé ou introduit, tous les incréments internes au bloc dans lequel se trouve cette déclaration sont considérés comme invalides. Dans le cas où l'on introduit une déclaration d'un identificateur, nous ne pouvons pas vérifier si cet identificateur est utilisé ou non dans le bloc. On rend donc les incréments de ce bloc invalides. S'il existe dans un bloc interne à ce bloc une autre déclaration de cet identificateur, les incréments de ce bloc restent valides.

Soit :



Pour cela il faut que l'on connaisse le numéro de l'incrément de début de bloc. A l'aide de l'ordre initial des incréments nous retrouvons l'incrément de début de bloc, ainsi que le numéro de ce bloc.

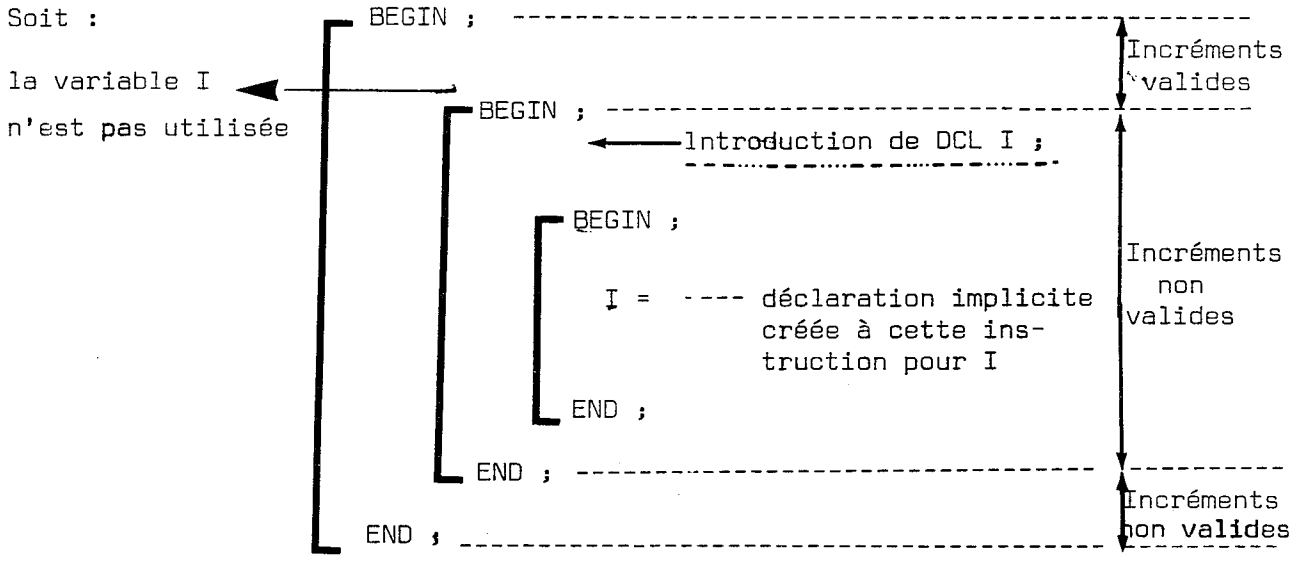
On cherche s'il existe un bloc, interne à ce bloc, dans lequel le même identificateur est déclaré. La table des symboles, la table des blocs et le chaînage entre les déclarations sont utilisés pour savoir s'il n'existe pas une telle déclaration dans un bloc interne.

a - Introduction d'une nouvelle déclaration

Il faut vérifier si une déclaration implicite a été créée pour cet identificateur valable pour ce bloc et les blocs internes (sont exclus les blocs internes dans lesquels cet identificateur est déclaré explicitement).

A l'aide de la table des symboles et du chaînage des déclarations de cet identificateur, on accède au dictionnaire d'attributs qui indique une déclaration implicite : pour un numéro de bloc égal à zéro. Si une telle déclaration existe, le dictionnaire d'attributs contient le numéro de bloc à partir duquel cette déclaration a été créée. Si ce bloc est le bloc de la nouvelle déclaration, ou un bloc interne, la déclaration implicite ainsi créée n'est plus valide. L'accès à cette déclaration dans le dictionnaire d'attributs est supprimé.

La portée d'une telle déclaration est le segment tout entier. On ne sait pas si cet identificateur est utilisé dans la suite du programme, on considère comme invalides les incréments de la suite du programme.



b - Suppression d'une déclaration

Il faut supprimer l'accès à cette déclaration dans le dictionnaire d'attributs. Le pseudo-code de cette déclaration, l'index (ou les index) des identificateurs, permettent de modifier la table des symboles et le chaînage entre les déclarations.

Phase 3 - Traitement des incréments IF-ELSE, DU-END

La modification de telles instructions ne change que le chaînage entre ces instructions (réalisé dans le dictionnaire de commande).

Phase 4 - Autres incréments

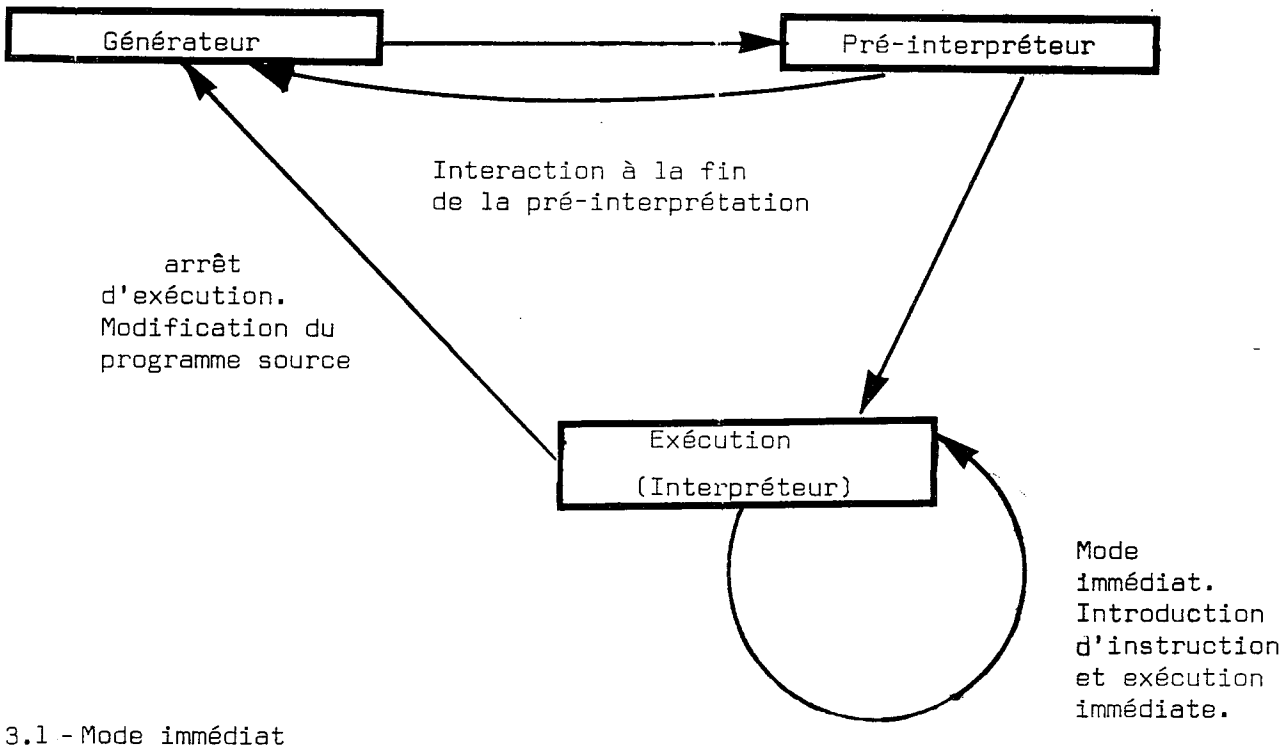
Les incréments d'un autre type ne sont plus valides. La pré-interprétation n'est reprise que pour les incréments non valides.

Pour éviter cette recherche des incréments non valides, on pourrait introduire de nouvelles commandes, plus particulièrement la commande de remplacement d'un incrément par un autre. Actuellement il faut utiliser successivement les commandes SKIP (et AFTER), et introduire le nouvel incrément ; on pourrait avoir ces deux informations en même temps par une commande du type

REPLACE 'numéro d'incrément' BY 'incrément' ;

III - INTERACTION PENDANT L'EXECUTION

Les interactions sont aussi possibles en cours d'exécution. Les flèches du graphe suivant indiquent les passages entre les différentes phases et les interactions possibles.



3.1 - Mode immédiat

L'exécution peut être arrêtée par :

- la commande WAIT
- le programmeur appuie sur la touche "attention" (l'exécution est arrêtée à la fin d'un incrément)
- une erreur.

Dans ce mode le programmeur peut entrer une introduction PL/1 qui est immédiatement exécutée. Les variables utilisées dans ces incréments doivent être valides à l'endroit du programme où a été effectuée l'interruption ; il n'y a aucune modification du texte source, et l'on utilise la pile d'exécution telle qu'elle existe ; il n'est alors pas possible d'introduire une nouvelle déclaration.

Dans ce mode il n'est pas intéressant d'utiliser la phase de pré-interprétation. En effet, ces instructions ne sont exécutées qu'une fois ; il est

superflu de les pré-interpréter : dans le mode immédiat on peut interpréter directement le pseudo-code fourni par le générateur (interpréteur ancienne version).

Nous pouvons d'autre part, dans ce mode, modifier le texte source, avec les possibilités d'édition

3.2 - Interaction pendant l'exécution avec édition

Lorsqu'on se trouve dans le mode immédiat, on peut modifier le programme initial. Ceci est effectué en utilisant la commande "SEGMENT *" . Les nouveaux incréments sont traités par le générateur de la même manière que ceux du segment initial et sont insérés dans ce segment. L'utilisateur peut aussi modifier des incréments existants.

L'édition,dans ce mode, est abandonné à l'aide de l'une des trois commandes suivantes :

a - RESTART qui reprend l'exécution au point d'arrêt.

b - EXECUTE qui recommence l'exécution de tout le segment.

c - GOTO Label ;

Puisqu'il y a insertion des modifications dans le texte source, la phase d'interprétation est reprise comme précédemment en analysant les incréments modifiés et en cherchant le nombre d'incréments à traiter dans cette phase.

A l'issue des modifications, si l'on reprend l'exécution par la commande EXECUTE, il n'apparaît aucun problèmes nouveaux, car toutes les modifications sont prises en compte.

Par contre, si l'on reprend l'exécution par la commande RESTART cela signifie que l'exécution continue en utilisant la pile d'exécution dans l'état où elle était au moment de l'arrêt. De même pour la troisième possibilité.

Les éléments de cette pile n'ont plus aucun sens si l'on effectue des modifications sur les incréments contenus dans des blocs actifs au moment de

l'arrêt de l'exécution. Nous sommes donc obligés d'analyser plus précisément les incréments introduits ou modifiés en fonction des conséquences qu'il peuvent avoir sur les éléments de la pile d'exécution.

1. L'insertion ou la suppression d'une déclaration dans un bloc actif obligerait à modifier la pile pour les éléments de ce bloc. Cette modification n'est pas prise en compte si on continue l'exécution à partir du point d'arrêt. De plus, du fait de la reprise de la pré-interprétation pour les incréments modifiés (ou rendus non valides pour la modification), la nouvelle représentation des identificateurs , dont on a introduit une déclaration, donne un déplacement non existant dans la pile. Dans un tel cas, il faut empêcher l'utilisateur de redémarrer l'exécution au point d'arrêt, ou à l'étiquette précisée.

Par contre, s'il s'agit d'une modification d'une déclaration d'un bloc non actif, l'utilisateur peut faire ce qu'il veut en prenant ses risques.

2. La modification d'un incrément BEGIN ou PROCEDURE et d'un END associé modifie la structure du segment et provoque la pré-interprétation de tout le segment et, par conséquent, rend non valide l'état actuel de la pile. L'utilisateur est prévenu de la non possibilité de redémarrer l'exécution au point d'arrêt.

3. S'il s'agit d'un autre incrément, seulement les valeurs peuvent avoir été modifiées, et à ce moment-là c'est au programmeur de choisir de quelle façon il désire reprendre l'exécution.

TROISIEME PARTIE

CHAPITRE I

FONCTIONNEMENT DE LA PILE D'INTERPRETATION

L'interpréteur utilise le dictionnaire de commande pour déterminer l'ordre lexicographique des incréments formant le segment à exécuter. Pour interpréter le nouveau pseudo-code, nous utilisons une pile pour allouer la mémoire aux variables.

I - ALLOCATION ET LIBERATION DE LA MEMOIRE -

1.1 - Allocation de mémoire

a. Variables statiques.

Les variables déclarées STATIC dans un segment sont reconnues avant l'interprétation. Nous avons l'information sur l'encombrement mémoire nécessaire pour ces variables, ce qui permet de réserver la place correspondante.

Si une telle variable est déclarée avec l'attribut INITIAL, l'initialisation est effectuée une seule fois, à la première activation du bloc contenant la déclaration (nous mettons un indicateur dans le dictionnaire d'attributs).

b. Variables automatiques.

Les variables de type AUTOMATIC sont allouées à chaque entrée dans le bloc contenant leurs déclarations.

Dans le nouveau pseudo-code, au numéro d'incrément correspondant à un début de bloc, est associée la liste des déclarations explicites de ce bloc. Cette liste donne, en particulier, les adresses des déclarations dans le dictionnaire d'attributs. Ainsi, connaissant les éléments constants de la déclaration et les éléments variables (dans le pseudo-code), on peut calculer l'espace de mémoire nécessaire à ces variables.

Les variables déclarées implicitement, sont mises en mémoire, à la suite des variables statiques.

1.2-Accès à la valeur d'une variable dans la pile -

Une référence à une variable est représentée par l'adresse dans le dictionnaire d'attributs correspondant à la déclaration valide pour le bloc concerné du programme.

Cette adresse permet de connaître :

a) le numéro du bloc contenant la déclaration de la variable.

b) le déplacement par rapport au début de la zone de mémoire(réservée aux variables AUTOMATIC du bloc) soit de la valeur de la variable, soit des fonctions d'accès à cette valeur (tableaux ....).

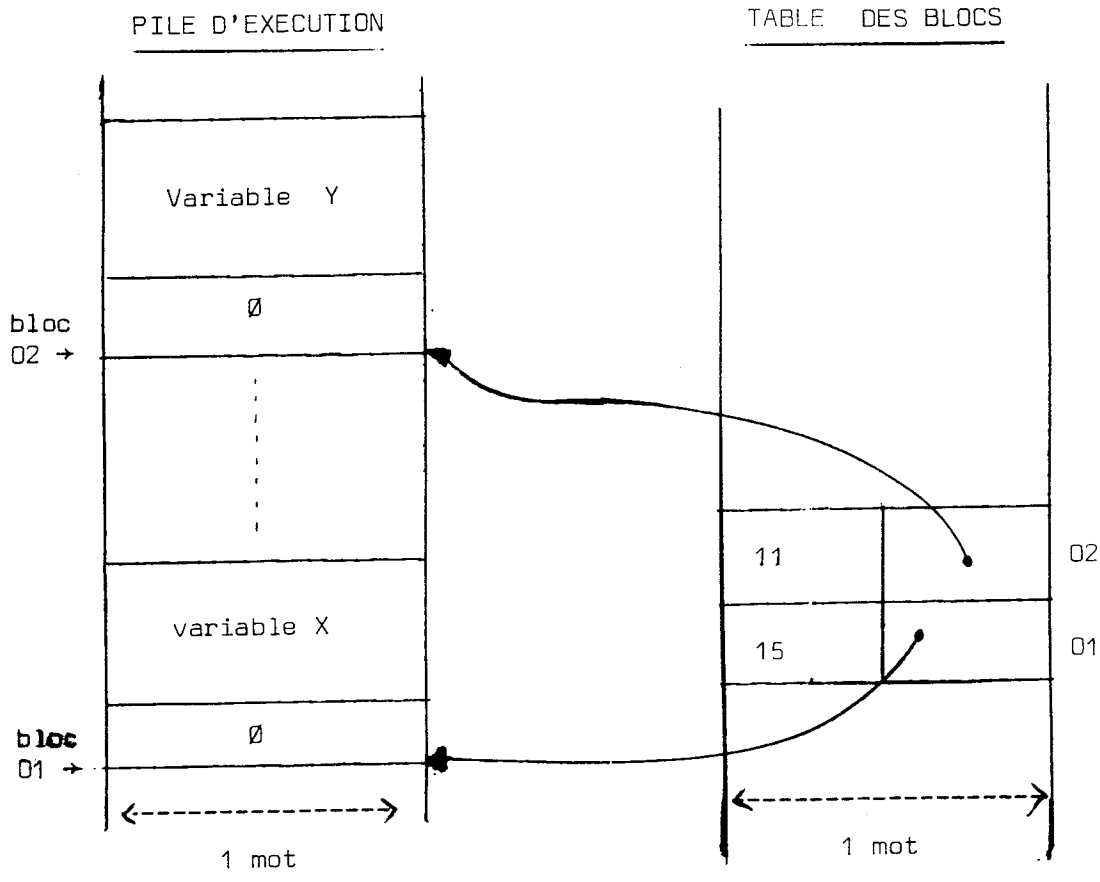
L'adresse de début de la zone de mémoire réservée au bloc est obtenue à l'aide d'une table des blocs actifs à un instant donné. A l'entrée de cette table qui correspond au numéro de bloc, on aura l'adresse du début de la mémoire allouée à ce bloc s'il est actif, et zéro s'il est désactivé.

Pour pouvoir plus aisément libérer la mémoire en fin de bloc, nous enregistrons dans la table le numéro d'incrément contenant l'instruction END.

Exemple -

```
bloc n° 1 .....   03    BEGIN ;
                  04    DCL X ;
                   :
bloc n° 2 .....   07    A : BEGIN ;
                  08    DCL Y ;
                   :
                  10    .....
                  11    END A ;
                  12
                   :
                  15    END ;
```

A l'incrément 10, la pile d'exécution et la table des blocs sont dans l'état suivant :



De plus, si un bloc est de nouveau activé avant que la précédente activation soit terminée (par l'appel de procédures récursives) la table des blocs contiendra dans l'entrée correspondant au numéro de ce bloc un pointeur vers cette nouvelle activation du bloc. Au début de l'espace de mémoire réservé à cette nouvelle activation, nous aurons un pointeur vers l'emplacement de mémoire allouée à la précédente activation de ce bloc (s'il n'existe pas d'autre activation, la valeur du pointeur est zéro).

Ce chaînage entre les différentes activations d'un même bloc nous permet une mise à jour de la table des blocs au fur et à mesure que les nouvelles générations de blocs sont désactivées.

1.3 -Cas particulier : les procédures récursives -

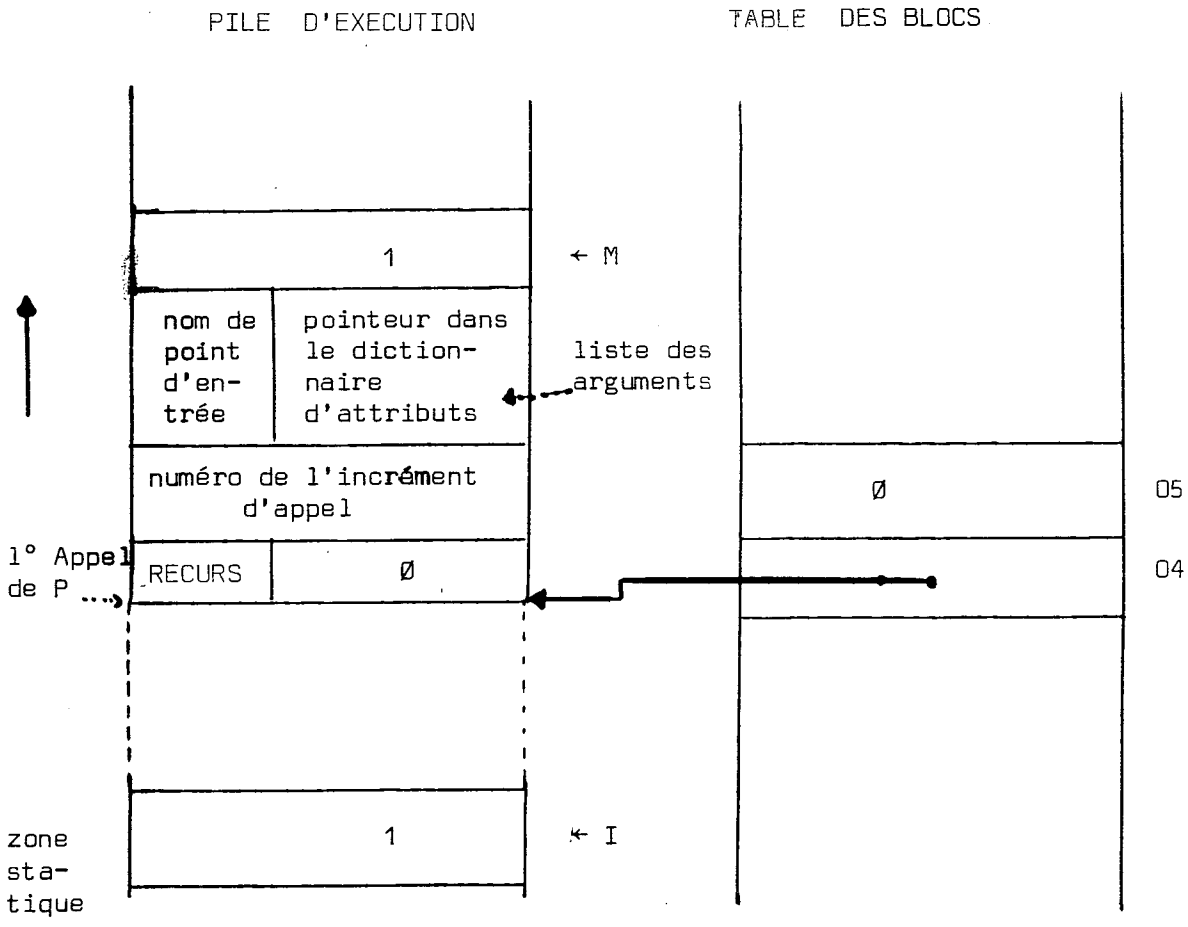
Nous allons voir sur un exemple, comment on traite de telles procédures.

```
bloc n° 4 ----> 006   P:PROCEDURE (Q) RECURSIVE ;
                007   DCL   (Q,R)  ENTRY , I STATIC  INITIAL (0) ,
                              M   AUTOMATIC ;
                008   I = I+1 ;
                009   M = I   ;
                010   LAB :  IF   I < 3   THEN
                011   CALL  P (R) ;
                012   ELSE
                013   CALL Q ;
                014   RETURN ;
bloc n° 5 ----> 015   R :  PROCEDURE ;
                016   PUT   LIST (M) ;
                017   RETURN ;
                018   END R ;
                019   END P ;
```

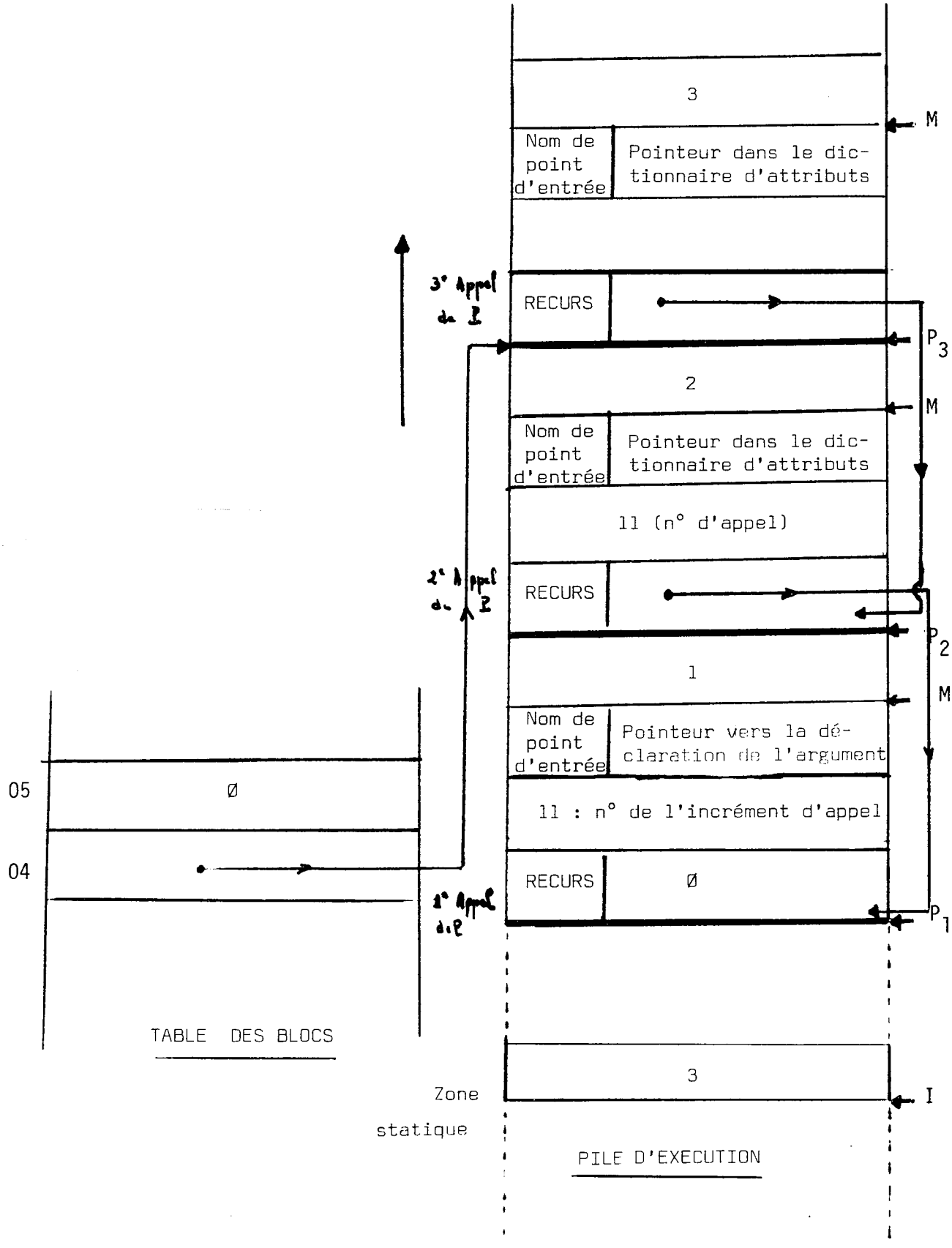
Au premier appel de la procédure P, la variable I (variable statique) est initialisée à zéro.

La valeur de M est 1 (exécution de l'incrément 9). L'exécution de l'incrément 10 (étiqueté LAB) provoque l'appel récursif de la procédure P avec, comme argument, la procédure R.

La pile et la table des blocs, juste avant le deuxième appel de P, SONT :

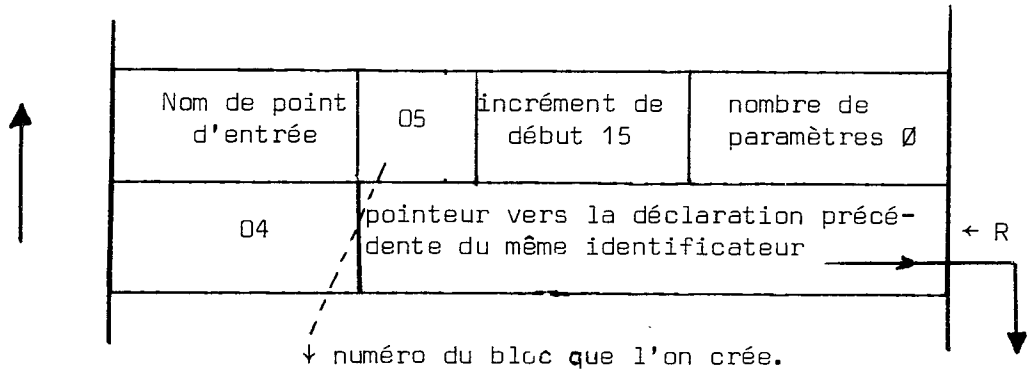


Au deuxième appel de la procédure, la valeur de la variable I est 2, et à l'incrément 11, on effectue un troisième appel de la procédure P avec comme argument R. La pile et la table des blocs sont alors dans l'état suivant :



Lors de ce troisième appel, la condition I < 3 n'est plus remplie, et l'on exécute l'incrément 13, c'est-à-dire que l'on appelle la procédure argument de la procédure P soit la procédure R.

Dans le dictionnaire d'attributs, on a sur cette procédure R, les informations suivantes :



R est une procédure dont le nom de point d'entrée est déclaré explicitement dans le bloc n° 4. Le bloc que l'on traite (ou bloc courant) est le bloc n° 4.

La procédure R étant un argument du troisième appel de la procédure P, sa déclaration doit être connue avant cette nouvelle activation du bloc 4.

Le bloc englobant le bloc 5 est la deuxième activation du bloc 4.

Nous avons, de la troisième activation du bloc 4, un pointeur vers le début de la deuxième activation du bloc 4.

A l'exécution de l'incrément 16 (PUT LIST (M)), la valeur de M imprimée est 2.

1.4-Désactivation des blocs -

Nous allons voir sur l'exemple précédent, comment on libère la mémoire allouée à chaque bloc.

L'exécution de l'incrément 14 termine la troisième activation de la procédure P. On met à jour la table des blocs actifs, en utilisant le chaînage entre les blocs n° 4 actifs.

L'entrée dans cette table, pour le bloc n° 4, contiendra un pointeur vers la deuxième activation de la procédure P (pointeur P2). L'on continue l'exécution de la procédure P et à l'exécution de l'incrément 14, cette deuxième génération de P est désactivée. L'on procède de nouveau à la mise à jour de la table des blocs.

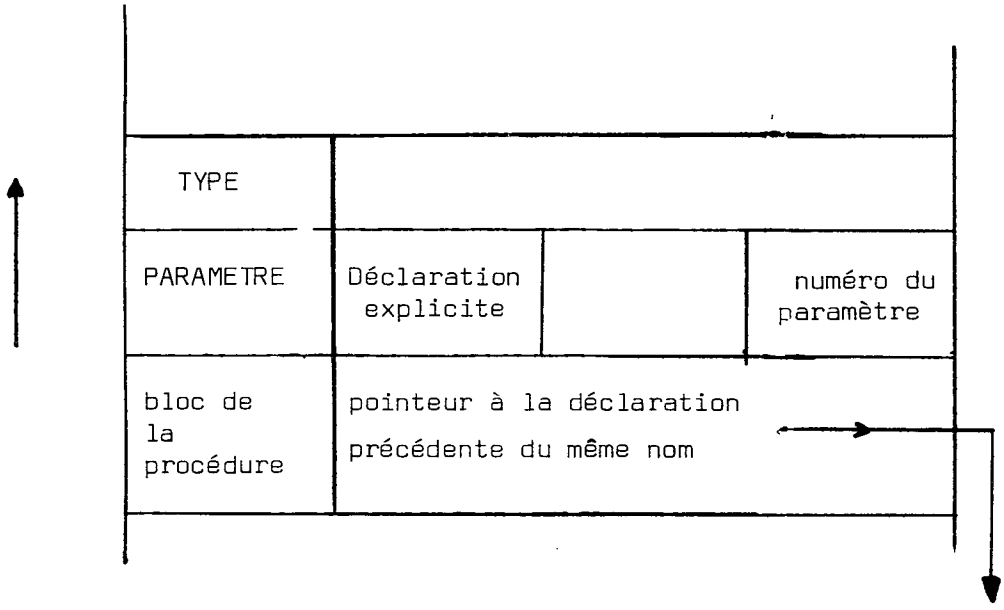
Lors de l'exécution suivante de l'incrément 14, on termine la première activation de P. La table contient alors, à l'entrée pour le bloc n° 4, la valeur 0.

II - INTERPRETATION DES INCREMENTS -

Nous avons vu que, pour les expressions, nous avions pour chaque identificateur suffisamment d'informations dans la fonction de conversion, pour appeler les routines adéquates et effectuer les conversions. L'exécution des opérations et des autres incréments se fait de façon analogue à celle effectuée par l'ancien interpréteur.

Nous préciserons simplement ici, comment s'effectue l'association paramètre-argument dans un appel de procédure.

Lors de la construction du dictionnaire d'attributs, nous avons les informations concernant les déclarations de paramètre, qui sont :



Dans le corps de procédure, nous avons effectué l'association utilisation du paramètre-déclaration du paramètre, c'est-à-dire que dans le nouveau pseudo-code, à l'utilisation du paramètre, nous aurons le déplacement dans le dictionnaire d'attributs de la déclaration du paramètre.

Nous avons ainsi le numéro du paramètre dans la liste de paramètres et sur la pile d'exécution nous avons la liste des pointeurs vers les déclarations des arguments dans l'ordre donné par l'appel de procédure.

CHAPITRE II

MESURE DU TEMPS D'EXECUTION

I - INTRODUCTION

Les deux points sur lesquels nous avons fait porter notre effort d'optimisation sont l'identification des variables et l'examen préalable des conversions à effectuer.

La phase de génération n'ayant pas été changée, nous mesurons seulement les temps d'exécution.

Nous désignons par :

- T_A le temps d'exécution de l'ancienne version,

- T_N le temps d'exécution de la nouvelle version qui se décompose en T_P temps nécessaire à la phase de pré-interprétation et T_I temps nécessaire à la phase d'interprétation.

Nous calculons :

- $\frac{T_A}{T_N}$ qui est le rapport des temps des deux versions. Il représente le facteur de gain de temps entre les deux versions.

- $\frac{T_A - T_N}{T_A}$ qui est le gain relatif.

Nous avons choisi pour les mesures des temps d'exécution un ensemble d'exemples comportant une boucle. Nous avons essayé de voir quelle est l'influence, d'une part, du nombre de variables intervenant dans les expressions de la boucle, d'autre part, du nombre de conversions à effectuer sur les valeurs de ces variables.

II - INFLUENCE SUR LES TEMPS D'EXECUTION DU NOMBRE DE VARIABLES DANS LA BOUCLE

Pour mesurer l'influence sur les temps d'exécution du nombre de variables dans la boucle, nous avons choisi des exemples où il n'y avait pas de conversions.

2.1 - Exemple 1 : Dans la boucle il y a deux références à une variable

```
SEGMENT i ;
BEGIN ;
DCL  i  FIXED DEC (5) ;
DCL  x  FIXED REAL DEL (5) ;
DCL  y  FIXED REAL DEC (6) ;
y = 1 ;
DO      i = 1     BY 1   TO   10 ;
y = y + 1
END ;
PUT LIST   (y) ;
END ; EX ;
```

Les mesures sont les suivantes :

TABLEAU I

longueur de la boucle	nbre d'incréments exécutés	T_A ancienne version	T_P pré-interprétation	T_I interprétation	$T_N=T_P+T_I$ nouvelle version	$\frac{T_A}{T_N}$	$\frac{T_A-T_N}{T_A}$
40	88	234	13	104	117	2	0,5
50	108	277	13	127	140	1,97	0,494
60	128	332	13	154	167	1,98	0,496
70	148	379	12	179	191	1,98	0,496
80	168	427	10	201	211	2,02	0,505
90	188	494	10	230	240	2,06	0,512
100	208	534	13	252	265	2,03	0,505
110	228	590	13	278	291	2,02	0,506
120	248	638	10	301	311	2,05	0,512
130	268	706	10	322	332	2,12	0,529
140	288	727	13	347	360	2,01	0,5048
150	308	783	10	368	378	2,07	0,517
160	328	839	10	394	404	2,07	0,518
170	348	898	11	427	438	2,05	0,512
180	368	955	10	443	453	2,10	0,525
190	388	985	9	489	498	1,97	0,5035
200	408	1041	10	507	517	2,01	0,5033

On constate que la facteur gain de temps $\frac{T_A}{T_N}$ augmente avec la longueur de la boucle et reste à peu près égal à 2.

2.2 - Exemple 2 : Dans la boucle il y a 4 références à une variable

Il n'y a toujours aucune conversion sur les variables. Quatre variables sont utilisées au lieu de 2 dans l'exemple précédent.

```
SEGMENT i ;
BEGIN ;
DCL  i  FIXED DEC (5) ;
DCL  v  FIXED DEC (6) ;
DCL  u  FIXED DEC (6) ;
DCL  j  FIXED DEC ;
v = 1 ; j = 1 ;
u = 1 ;
DO     i = 1     TO     40     BY 1 ;
u = v + j + u ;
END ;
END ; EX ;
```

Les résultats des mesures sont donnés dans le tableau II suivant.
Nous constatons que l'on atteint un gain 2 plus rapidement que précédemment (longueur de boucle 40) lorsque le nombre de variables augmente.
Ces résultats sont comparés sur un graphique (Figure I).

TABLEAU II

longueur de la boucle	nbre d'incréments exécutés	T_P pré-interprétation	T_I interprétation	$T_N = T_P + T_I$ nouvelle version	T_A ancienne version	$\frac{T_A}{T_N}$	$\frac{T_A - T_N}{T_A}$
40	90	12	108	120	268	2,23	0,55
50	110	12	152	164	339	2,06	0,516
60	130	12	182	194	396	2,04	0,510
70	150	14	216	230	450	1,95	0,488
80	170	15	235	250	514	2,056	0,513
90	190	13	266	279	571	2,046	0,511
100	210	16	290	306	635	2,075	0,518
110	230	15	314	329	685	2,082	0,519
120	250	16	325	341	744	2,18	0,541
130	270	14	399	413	813	1,968	0,492
140	290	13	409	422	874	2,07	0,517
150	310	15	440	455	921	2,02	0,505
160	330	13	484	497	980	1,97	0,492
170	350	16	497	513	1033	2,01	0,50
180	370	14	524	538	1112	2,066	0,516
190	390	15	560	575	1175	2,043	0,510
200	410	12	615	627	1256	2,003	0,500
210	430	16	588	604	1294	2,14	0,533

III - INFLUENCE SUR LES TEMPS D'EXECUTION DU NOMBRE DE CONVERSIONS A EFFECTUER SUR LES VARIABLES DE LA BOUCLE

Nous voulons voir si l'examen préalable des conversions est un facteur intervenant dans le gain de temps.

3.1 - Exemple 3 : Existence de conversions dans la boucle (2 conversions)

Dans l'expression u = v + j, la valeur de v doit être convertie de décimale en binaire, le résultat v + j doit être ensuite converti de binaire à décimale.

```
      SEGMENT i ;
      BEGIN ;
      DCL  i  FIXED DEC (5) ;
      DCL  v  FIXED DEC (6) ;
      DCL  u  DEC FIXED (6)
      DCL  j  FIXED BIN ;
      v = 1 ; j = 1 ;
      DO   i = 1  BY  1  TO  70 ;
      u = v + j ;
      END ;
      END ; EX ;
```

Les résultats sont donnés dans le tableau III suivant. L'amélioration diminue et reste inférieure à un facteur 2 ; le facteur gain de temps est de l'ordre de 1,92 quelque soit la longueur de la boucle.

TABLEAU III

longueur de la boucle	nbre d'incréments exécutés	T_P pré-interprétation	T_I interprétation	$T_N=T_P+T_I$	T_A ancienne version	$\frac{T_A}{T_N}$	$\frac{T_A-T_N}{T_A}$
40	90	15	120	135	257	1,90	0,474
50	110	15	150	165	318	1,92	0,48
60	130	12	184	196	366	1,86	0,464
70	150	15	209	224	424	1,89	0,471
80	170	12	236	248	487	1,96	0,490
90	190	12	273	285	533	1,87	0,465
100	210	15	290	305	593	1,94	0,485
110	230	15	319	334	651	1,94	0,486
120	250	12	357	369	724	1,96	0,49
130	270	12	386	398	762	1,91	0,477
140	290	13	410	423	849	2,00	0,501
150	310	14	437	451	883	1,95	0,489
160	330	11	473	484	954	1,97	0,492
170	350	12	503	515	1005	1,95	0,487
180	370	15	532	547	1043	1,90	0,475
190	390	15	550	565	1107	1,95	0,489
200	410	12	592	604	1177	1,94	0,486
210	430	12	628	640	1215	1,89	0,473

3.2 - Exemple 4 : Existence de trois conversions dans la boucle

```
SEGMENT i ;
BEGIN ;
DCL   i   FIXED DEC (5) ;
DCL   v   FIXED DEC (6) ;
DCL   u   DEC FIXED (6) ;
DCL   j   FIXED BIN ;
v = 1 ; j = 1 ;
u = 1 ;
DO    i = 1   TO   120   BY 1 ;
u  = v + j + u ;
END ;
END ; EX ;
```

```
***DEBUT D'EXECUTION...
TEMPS DE PREINTERPRETATION =0000000013  MILLISECONDES
TEMPS D'EXECUTION =0000000421  MILLISECONDES
 ***FIN D'EXECUTION- NB D'INCR=0000000250
```

u = v + j + u, la valeur de u est convertie de décimale en binaire, la valeur de u est convertie de décimale en binaire, la valeur de v + j + u est convertie de binaire en décimale.

Les résultats sont donnés dans le tableau IV.

TABLEAU IV

longueur de la boucle	nbre d'incréments exécutés	T_P pré-interprétation	T_I interprétation	$T_N=T_P+T_I$	T_A ancienne version	$\frac{T_A}{T_N}$	$\frac{T_A-T_N}{T_A}$
40	90	15	128	143	295	2,06	0,515
50	110	16	152	168	361	2,14	0,534
60	130	16	211	227	437	1,91	0,480
70	150	13	247	260	496	1,90	0,475
80	170	16	258	274	568	2,07	0,517
90	190	16	298	314	627	1,99	0,499
100	210	12	349	361	708	1,96	0,490
110	230	16	398	414	776	1,87	0,466
120	250	13	421	434	845	1,94	0,486
130	270	16	451	467	919	1,96	0,491
140	290	16	478	494	982	1,98	0,496
150	310	14	533	547	1033	1,88	0,470
160	330	13	554	567	1106	1,95	0,487
170	350	16	615	631	1179	1,81	0,464
180	370	16	637	653	1238	1,89	0,472
190	390	16	665	681	1330	1,95	0,487
200	410	12	721	733	1378	1,87	0,468
210	430	16	730	746	1442	1,93	0,482

3.3 - Exemple 5 : Existence de quatre conversions dans la boucle

```
SEGMENT i ;
BEGIN ;
DCL   i   FIXED DEC (5) ;
DCL   v   FIXED DEC (6) ;
DCL   u   DEC FLOAT (6) ;
DCL   j   FIXED BIN ;
v = 1 ; j = 1 ; u = 1 ;
DO     i = 1    TO 40   BY 1 ;
y = v + j + u ;
END ;
END ; Ex ;
```

Dans l'expression se trouvant dans la boucle DO comprend 4 conversions : la valeur de v est convertie de décimal en binaire, la valeur de u est convertie de décimal en binaire, la valeur du résultat intermédiaire v + j est convertie de fixe en flottant, et le résultat v + j + u qui est de type flottant binaire est converti de binaire en décimale.

Les résultats sont donnés dans le tableau V suivant.
Nous constatons que le facteur gain de temps diminue avec le nombre de conversions, il est seulement de l'ordre de 1,9.

TABLEAU V

longueur de la boucle	nbre d'incréments exécutés	T_P pré-interprétation	T_I interprétation	$T_N=T_P+T_I$	T_A ancienne version	$\frac{T_A}{T_N}$	$\frac{T_A-T_N}{T_A}$
40	90	13	149	162	292	1,802	0,445
50	110	17	183	200	359	1,795	0,442
60	130	14	223	237	427	1,80	0,444
70	150	13	256	269	489	1,81	0,449
80	170	13	290	303	565	1,86	0,463
90	190	13	325	338	630	1,86	0,463
100	210	13	363	376	696	1,85	0,459
110	230	13	393	406	759	1,86	0,465
120	250	13	443	456	834	1,82	0,453
130	270	14	468	482	922	1,91	0,477
140	290	13	512	525	983	1,87	0,465
150	310	13	541	554	1035	1,90	0,474
160	330	15	566	581	1095	1,88	0,469
170	350	15	605	620	1189	1,91	0,4785
180	370	13	638	651	1268	1,94	0,4865
190	390	16	692	708	1317	1,86	0,4624
200	410	13	720	733	1387	1,89	0,4715
210	430	13	760	773	1477	1,91	0,4766

IV - INFLUENCE SUR LES TEMPS D'EXECUTION DE LA NATURE DES CONVERSIONS A EFFECTUER

Nous effectuons ces mesures sur deux exemples pour des longueurs de boucles différentes, d'une part une longueur 5, d'autre part une longueur 60.

4.1 - Exemple 6 : une seule conversion dans une boucle de longueur 60

```
SEGMENT i ;
BEGIN ;
DCL  i ;
DCL  x   FIXED DEC (6) ;
DCL  y   FLOAT BIN (6) ;
y = 1 ;
DO   i = 1   TO 60;
x = y ;
END ;
PUT LIST (x) ;
END ;
```

On fait varier les types de x et y. Les résultats sont groupés dans le tableau VI suivant.

Il n'y a dans la boucle qu'une conversion à effectuer et dans certains cas, il y a une conversion pour la variable x intervenant dans l'instruction PUT LIST.

TABLEAU VI

TYPE de x	TYPE de y	conversion pour x = y	T_P pré-interprétation	T_I interprétation	$T_N = T_P + T_I$	T_A ancienne version	$\frac{T_A}{T_N}$	$\frac{T_A - T_N}{T_A}$
Fixed decimal	Fixed decimal	aucune	10	102	112	252	2,25	0,50
Fixed decimal	Fixed binary	BIN →DEC	10	97	107	278	2,59	0,61
Fixed decimal	Float binary	BIN →DEC FLOAT →FIXED	10	117	127	288	2,26	0,55
Fixed decimal	Float decimal	FLOAT →FIXED	10	125	135	288	2,13	0,53
Float decimal	Float decimal	aucune	10	96	106	259	2,44	0,59
Float decimal	Float binary	BIN →DEC	10	98	108	272	2,51	0,60
Float decimal	Fixed binary	BIN →DEC FIXED →FLOAT	10	102	112	284	2,53	0,60
Float decimal	Fixed decimal	FIXED →FLOAT	9	106	115	276	2,4	0,58
Float binary	Fixed decimal	DEC →BIN FIXED →FLOAT	11	119	130	287	2,20	0,547
Float binary	Fixed binary	FIXED →FLOAT	9	104	113	273	2,41	0,586
Float binary	Float binary	aucune	11	100	111	256	2,3	0,562
Float binary	Float decimal	DEC →BIN	10	113	123	271	2,20	0,546
Fixed binary	Fixed binary	aucune	10	98	108	256	2,37	0,57
Fixed binary	Fixed decimal	DEC →BIN	11	114	125	278	2,224	0,55
Fixed binary	Float binary	FLOAT →FIXED	11	107	118	278	2,35	0,575
Fixed binary	Float decimal	DEC →BIN FLOAT →FIXED	11	115	126	279	2,21	0,548

Exemple 7 : Une seule conversion dans la boucle de longueur 120

```
SEGMENT  i;
BEGIN;
DCL   i;
DCL   X FIXED DEC (6);
DCL   Y FLOAT BIN (6);
Y=1;
DO i=1      TO 120;
X=Y;
END;
END;
EX;
```

Nous effectuons les mesures en faisant varier les types de x et y et nous obtenons les résultats suivants pour les 247 instructions exécutées. Les résultats sont groupés dans le tableau VII suivant.

Les exemples 6 et 7 montrent que les gains sont plus élevés lorsqu'il n'y a pas de conversion à effectuer. Les gains sont différenciés selon le type de conversions. Pour un type de conversion donné, on a la même évolution des gains entre les deux exemples.

De plus, si dans l'instruction y=1, on exprime la constante selon le type de y, les résultats sont pratiquement identiques à ceux indiqués dans les tableaux VI et VII.

TABLEAU VII

TYPE de x	TYPE de y	conversion pour x = y	T_p pré-interprétation	T_I interprétation	$T_N = T_p + T_I$	T_A ancienne version	$\frac{T_A}{T_N}$
Fixed decimal	Fixed decimal	aucune	11	207	218	507	2,32
Fixed decimal	Fixed binary	BIN → DEC	10	241	251	530	2,11
Fixed decimal	Float binary	BIN → DEC FLOAT → FIXED	10	253	263	564	2,14
Fixed decimal	Float decimal	FLOAT → FIXED	10	258	268	553	2,06
Float decimal	Float decimal	aucune	10	181	191	484	2,53
Float decimal	Float binary	BIN → DEC	10	213	223	510	2,28
Float decimal	Fixed binary	BIN → DEC FIXED → FLOAT	10	237	247	530	2,14
Float decimal	Fixed decimal	FIXED → FLOAT	11	197	208	530	2,54
Float binary	Fixed decimal	DEC → BIN FIXED → FLOAT	10	199	209	543	2,59
Float binary	Fixed binary	FIXED → FLOAT	7	225	232	524	2,24
Float binary	Float binary	aucune	10	187	197	489	2,48
Float binary	Float decimal	DEC → BIN	10	209	219	504	2,30
Fixed binary	Fixed binary	aucune	7	175	185	484	2,61
Fixed binary	Fixed decimal	DEC → BIN	11	204	215	528	2,45
Fixed binary	Float binary	FLOAT → FIXED	10	217	227	516	2,27
Fixed binary	Float decimal	DEC → BIN FLOAT → FIXED	10	239	249	523	2,10

V - CONCLUSIONS

Si nous comparons les résultats obtenus sur l'ensemble des exemples donnés, nous constatons que le gain de temps $(\frac{T_A}{T_N})$ augmente :

- d'une part, avec le nombre d'expressions ou plus exactement avec le nombre de références aux variables existant dans la boucle (voir FIGURE I pour les exemples 1 et2, et FIGURE IV, pour les exemples 8 et 9),

- d'autre part, avec le nombre de conversions à effectuer sur les variables intervenant dans la boucle (voir FIGURE II, pour les exemples 4 et 5).

Contrairement à ce que nous avons pensé, le gain de temps, lorsque les expressions comportent des conversions est plus faible que lorsqu'elles n'en comportent pas. Nous avons comparé sur des graphiques les résultats obtenus lorsque pour un même exemple, on fait varier le type des variables afin qu'apparaissent des conversions dans la boucle (voir FIGURE III, pour les exemples 2 et 5, et FIGURE V pour les exemples 8 et 10).

Ces résultats signifient que le temps nécessaire pour effectuer les tests sur les types des variables est faible devant le temps nécessaire pour effectuer les conversions sur les valeurs. Le gain essentiel de temps est réalisé par l'association "utilisation-déclaration" des variables.

ANNEXE AU CHAPITRE II

Exemple 8. Influence du nombre de variables dans la boucle sur les mesures du temps (3 variables)

```
SEGMENT x ;
BEGIN ;
x = 1 ; DCL x,n,m ;
DCL bDEC FIXED (5,0) ;
n = 0 ; m = 4 ; b = 9 ;
x = (b + m) / (n + m) ;
PUT LIST (x) ;
BEGIN ;
k = x + b ;
END ;
BEGIN ;
DCL u DEC FLOAT (6) ;
DCL v DEC FLOAT (6) INITIAL (4) ;
DCL i INITIAL (6) ;
DCL y ;
y = 1 ;
DO i = 1 TO 60 ;
y = x   y ;
END ;
PUT LIST (y) ;
u = v + i + (x   y) ;
PUT LIST (u) ;
PUT LIST (k) ;
END ;
DCL k FLOAT DEC (b) ;
END ;
EX ;
ENDPLI ;
```

TABLEAU VIII

longueur de la boucle	Nombre d'incréments exécutés	T_A ancienne version	T_N nouvelle version	$\frac{T_A}{T_N}$	$\frac{T_A - T_N}{T_A}$
10	46	116	83	1,39	0,28
20	66	158	97	1,63	0,386
30	86	206	118	1,745	0,42
40	106	255	141	1,80	0,447
50	126	295	136	1,85	0,46
70	166	392	202	1,94	0,484
80	186	431	218	1,977	0,494
90	206	474	238	1,99	0,497
100	226	524	259	2,02	0,50

Exemple 9. Influence du nombre de variables dans la boucle sur les mesures du temps (11 références)

```
SEGMENT x ;
BEGIN ;
x = 1 ;
DCL x FLOAT DEC (6) ;
DCL n FLOAT DEC (6) ;
DCL m FLOAT DEC (6) ;
DCL bDEC FLOAT (6) ;
n = 0 ; m = 4 ; b = 9 ;
x = (b + m) / (n + m) ;
PUT LIST (x) ;
BEGIN ;
k = x + b ;
END ;
BEGIN ;
DCL u DEC FLOAT (6) ;
DCL v DEC FLOAT (6) INITIAL (4) ;
DCL j DEC FLOAT (6) INITIAL (6) ;
DCL y DEC FLOAT (6) ;
y = 1 ;
y = x * y ;
PUT LIST (y) ;
DCL i ;
DO i = 1 TO 40 ;
u = v + j + (x * y) ;
v = j ;
v = k + j ;
END ;
PUT LIST (v) ;
PUT LIST (u) ;
PUT LIST (k) ;
END ;
DCL k FLOAT DEC (6) ;
END ;
EX ;
```

```
***DEBUT D'EXECUTION...
TEMPS DE PREINTERPRETATION =0000000022  MILLISECONDES
 3.25000E+00
 3.25000E+00
 1.82500E+01
 3.48125E+01
 1.22500E+01
TEMPS D'EXECUTION = 0000000195  MILLISECONDES
 ***FIN D'EXECUTION- NB D'INCR=0000000190
```

Sur le graphe, nous constatons que les gains augmentent avec le nombre de variables. Dans l'exemple précédent, on atteint un gain d'un facteur 2 pour une boucle 100 alors que dans l'exemple 9, le gain atteint 2 dès que le boucle a une longueur 50.

TABLEAU IX

longueur de la boucle	Nombre d'incréments exécutés	T_A	T_N	$\frac{T_A}{T_N}$	$\frac{T_A-T_N}{T_A}$
10	70	173	108	1,6	0,375
20	110	269	149	1,805	0,446
30	150	376	191	1,968	0,492
40	190	464	234	1,982	0,495
50	230	562	281	2	0,50
60	270	663	329	2,015	0,503
70	310	732	365	2,00	0,501
80	350	852	398	2,14	0,532
90	390	956	440	2,172	0,539
100	430	1057	486	2,174	0,540
110	470	1177	534	2,204	0,5463
120	510	1303	588	2,2159	0,5487
130	550	1390	627	2,216	0,5489
140	590	1474	650	2,26	0,559

2. Exemple 2 comprenant des conversions

Nous modifions l'exemple précédent pour que dans les expressions de boucle interviennent des conversions. Pour cela, nous modifions le type des variables.

Exemple 10. Influence sur les temps d'exécution du nombre de conversions dans la boucle (8 conversions)

```
SEGMENT x ;
BEGIN ;
x = 1 ; DCL x,n,m ;
DCL bDEC FIXED (5,0) ;
n = 0 ; m = 4 ; b = 9 ;
x = (b + m) / (n + m) ;
PUT LIST (x) ;
BEGIN ;
k = x + b ;
END ;
BEGIN ;
DCL u DEC FLOAT (6) ;
DCL v DEC FLOAT (6) INITIAL (4) ;
DCL i INITIAL (6) ;
DCL y ;
y = 1 ;
y = x * y ;
PUT LIST (y) ;
DO i = 1 TO 10 ;
u = v + i + (x * y) ;
v = i ;
v = k + i ;
END ;
PUT LIST (v) ;
PUT LIST (u) ;
PUT LIST (k) ;
END ;
DCL k FLOAT DEC (6)
END ;
EX ;
```

Considérons les expressions intervenant dans le boucle DO, soit l'incrément :

u = u + i + (x * y)

Pour effectuer l'opération v + i, la valeur de u doit être convertie de décimale en binaire, la valeur de i de représentation fixe en flottante.

Dans l'opération x * y, il n'y a aucune conversion à effectuer.

Pour exécuter l'opération v + i + (x * y) le résultat intermédiaire x * y doit être converti de décimale en binaire.

Pour affecter la valeur de l'expression à la variable u, il faut convertir le résultat final de binaire en décimale.

Pour l'incrément :

u = i ;

La valeur i sera convertie de fine en flottant et de binaire en décimale.

Pour l'incrément :

u = k + i ;

Le résultat final est converti de fine en flottant et de binaire en décimale.

Pour l'ensemble de ces expressions, le nombre de conversions à effectuer est de 8.

Les résultats sont donnés dans le tableau 10.

TABLEAU X

longueur de la boucle	Nombre d'incréments exécutés	T_A	T_N	$\frac{T_A}{T_N}$	$\frac{T_A - T_N}{T_A}$
10	67	184	119	1,54	0,35
20	107	299	174	1,71	0,41
30	147	422	229	1,84	0,457
40	187	516	279	1,849	0,459
50	227	627	334	1,877	0,467
60	267	753	383	1,966	0,491
70	307	854	441	1,936	0,483
80	317	987	506	1,99	0,487
90	387	1072	556	1,92	0,481
100	427	1193	606	1,968	0,492
110	467	1325	661	2,00	0,50
120	507	1421	723	1,965	0,491
130	547	1533	785	1,95	0,48
140	587	1642	805	2,03	0,509

Exemple 11. Augmentation du nombre de conversions dans la boucle

Augmentons dans la boucle, le nombre de conversions à effectuer sur les valeurs. Par exemple, si nous déclarons la variable u de type FIXED DEC (5,0), alors dans l'expression :

```
u = v + i + (x * y)
```

le résultat final avant l'affectation à la variable u doit être aussi converti de float en fixed. Ce qui fait au total 9 conversions sur les valeurs des variables intervenant dans la boucle.

Les résultats sont les suivants :

TABLEAU XI

longueur de la boucle	Nombre d'incréments exécutés	T_A	T_N	$\frac{T_A}{T_N}$	$\frac{T_A - T_N}{T_A}$
40	230	601	310	1,938	0,484
80	430	1141	550	2,074	0,517
120	630	1668	790	2,111	0,5263
140	730	1939	917	2,114	0,5270

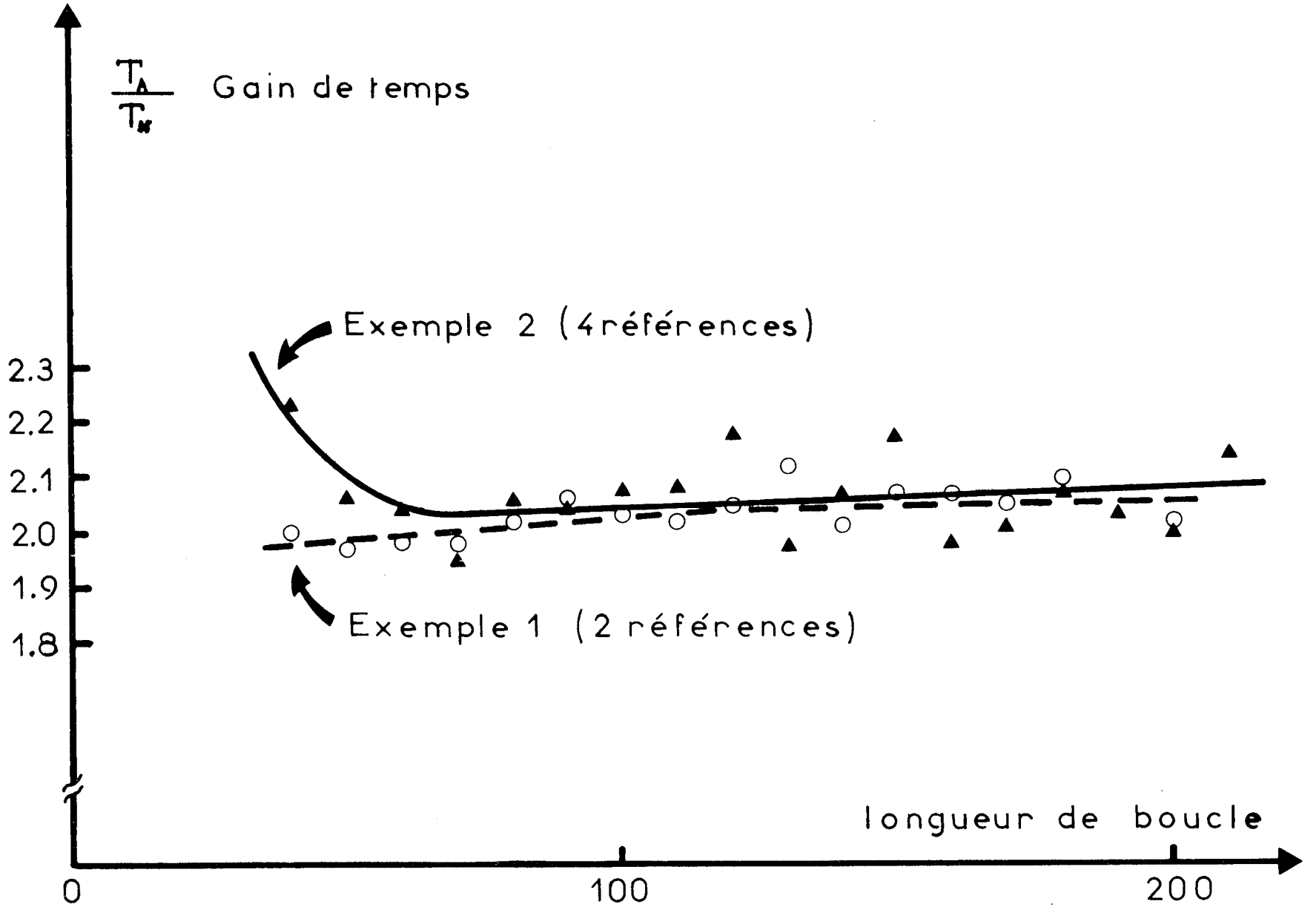

$\frac{T_A}{T_M}$ Gain de temps
Exemple 2 (4 références)
Exemple 1 (2 références)
2.3
2.2
2.1
2.0
1.9
1.8
longueur de boucle
0
100
200

Figure I

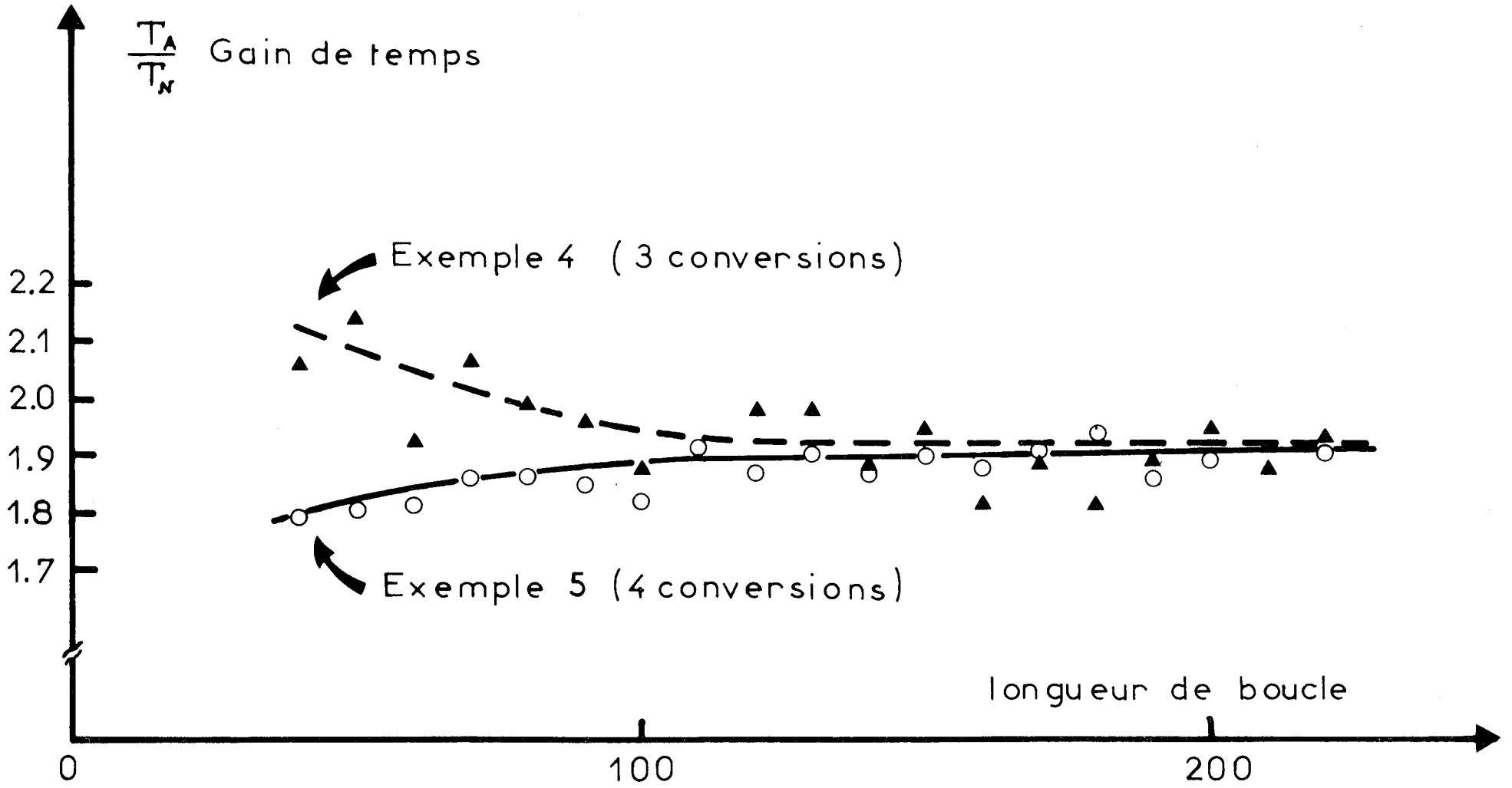

$\frac{T_A}{T_N}$ Gain de temps
Exemple 4 (3 conversions)
2.2
2.1
2.0
1.9
1.8
1.7
Exemple 5 (4 conversions)
longueur de boucle
0
100
200

Figure II

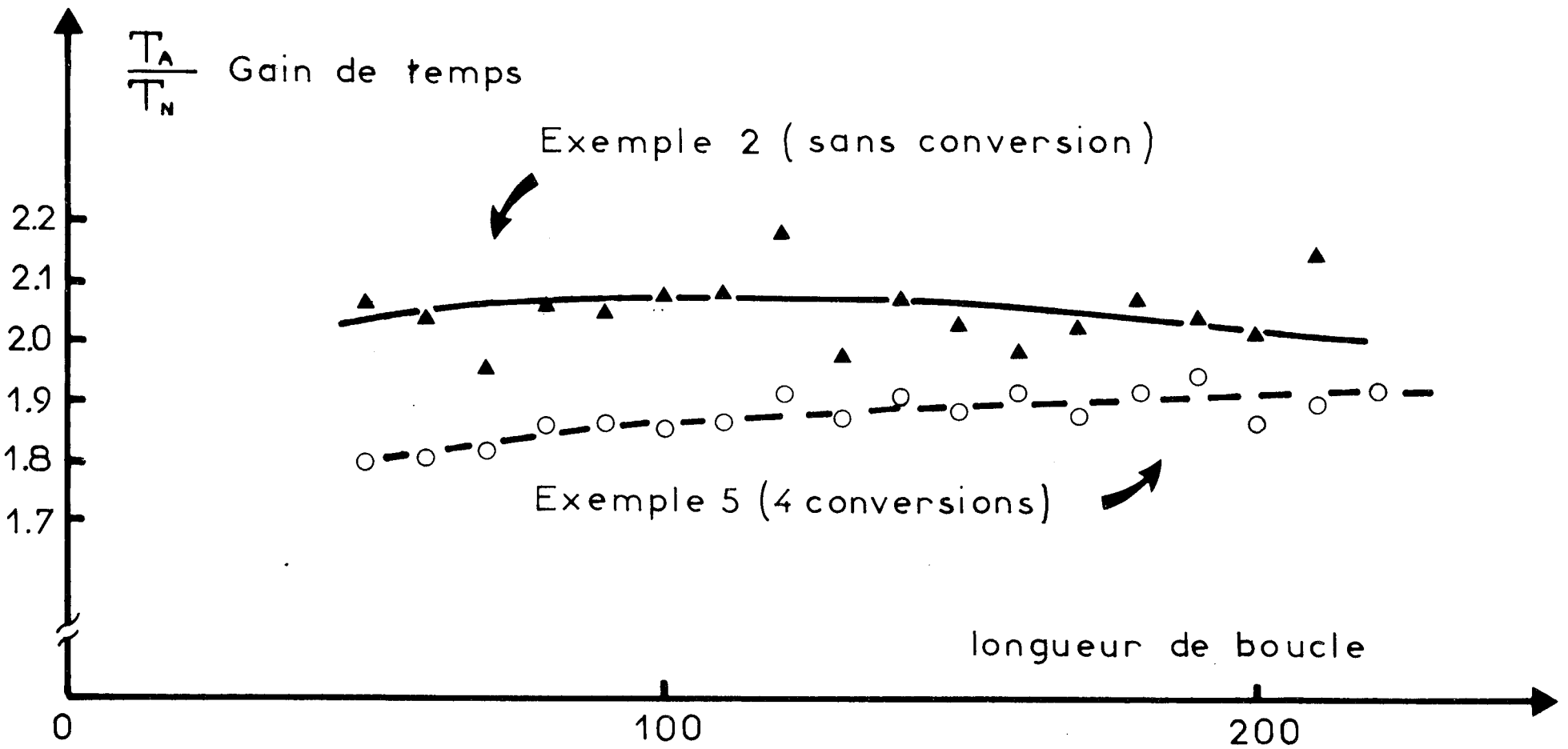

$\frac{T_A}{T_N}$ Gain de temps
Exemple 2 (sans conversion)
2.2
2.1
2.0
1.9
1.8
1.7
Exemple 5 (4 conversions)
longueur de boucle
0
100
200

Figure III

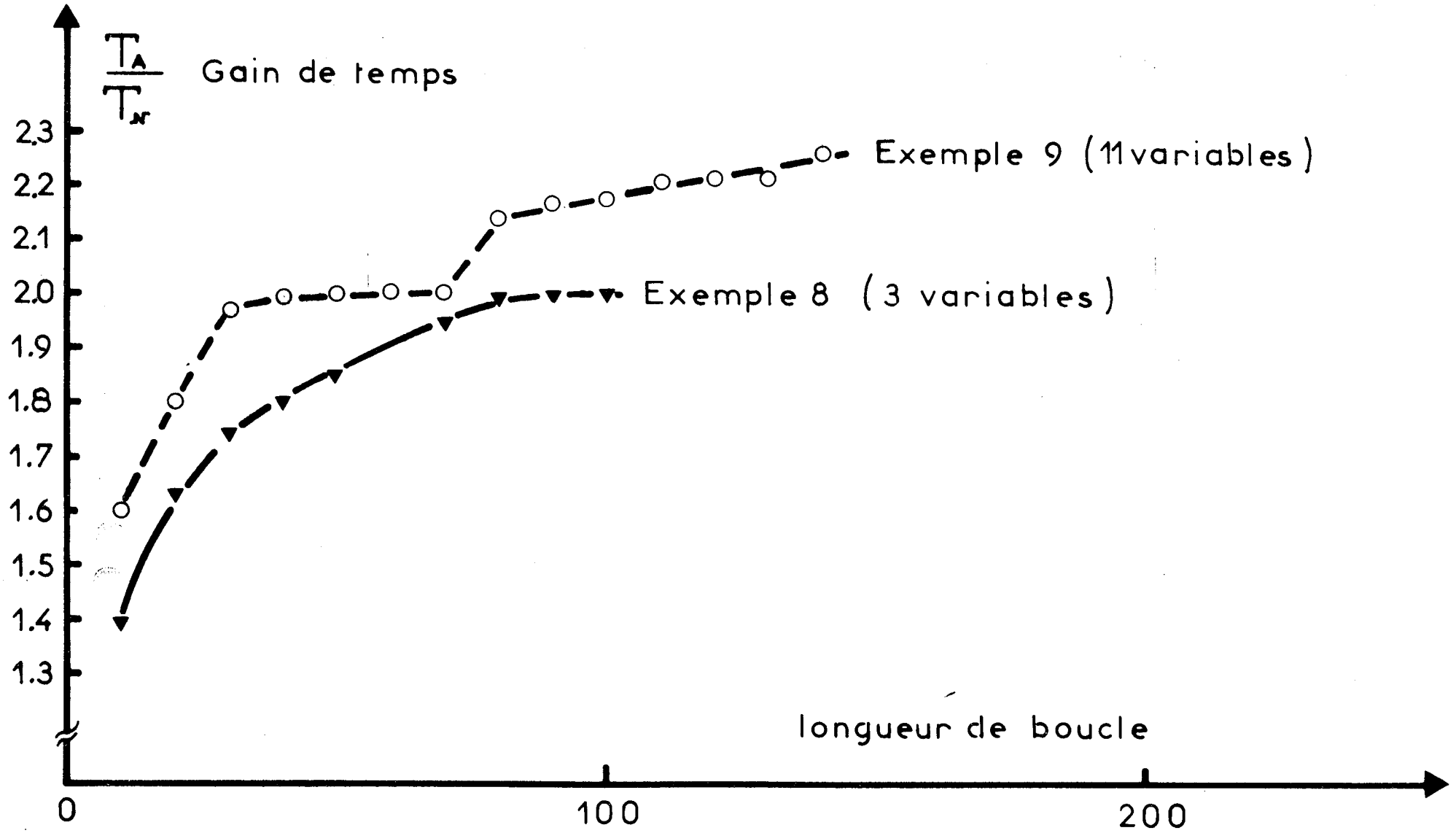

$\frac{T_A}{T_{NF}}$ Gain de temps
2.3
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
1.4
1.3
Exemple 9 (11 variables)
Exemple 8 (3 variables)
longueur de boucle
0
100
200

Figure IV

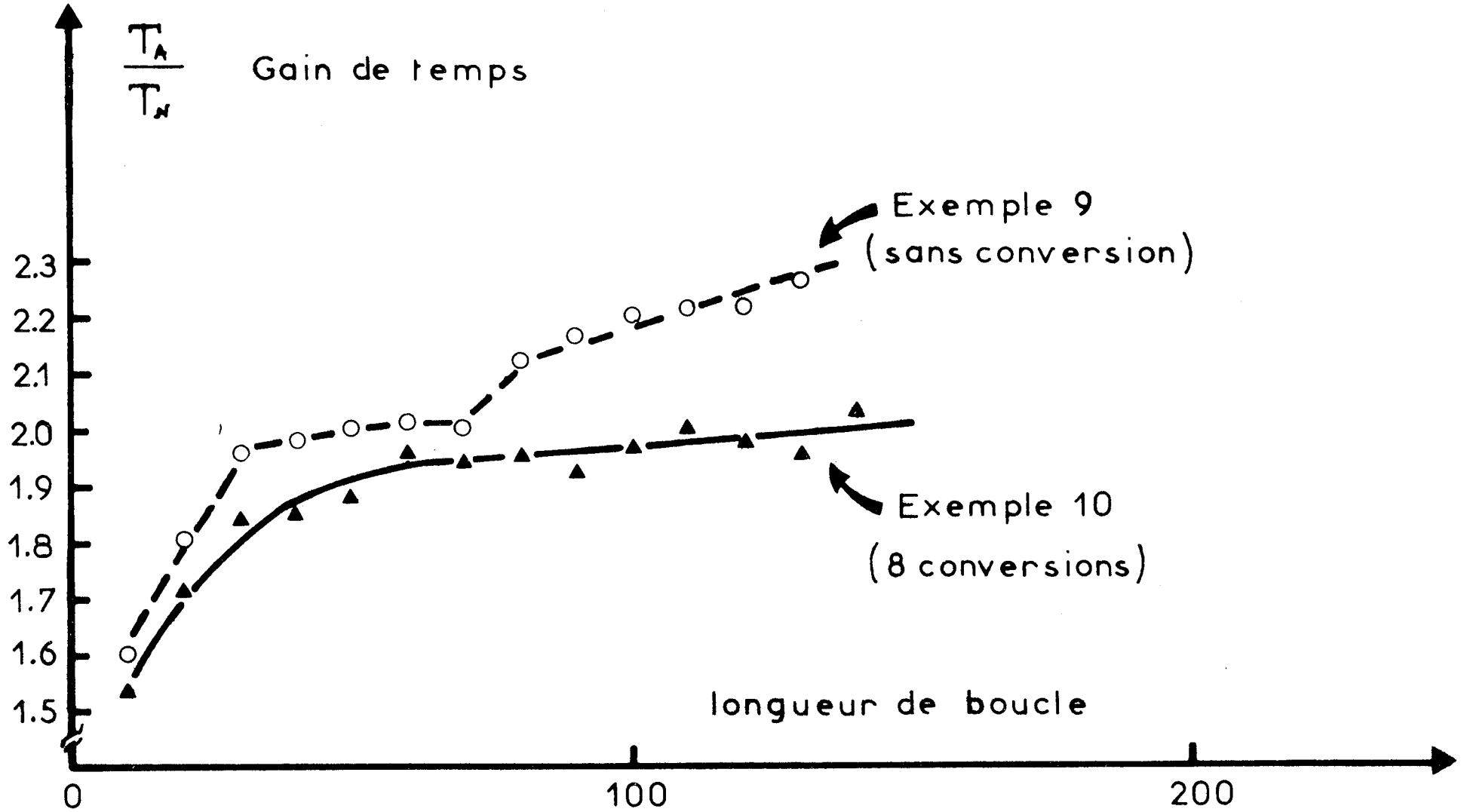

$\frac{T_A}{T_N}$
Gain de temps
Exemple 9
(sans conversion)
Exemple 10
(8 conversions)
2.3
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
longueur de boucle
0
100
200

Figure V

CHAPITRE III

CONCLUSION

Ce travail nous a permis de faire une analyse très précise des différentes fonctions du compilateur. En particulier, nous avons essayé de distinguer les éléments statiques et dynamiques. Cette distinction permet de construire le dictionnaire d'attributs afin d'effectuer, en particulier, l'association "utilisation-déclaration" des variables avant l'exécution, ce qui est applicable à tout autre langage (de même type global) pour la construction d'un compilateur interprétatif.

Dans cette version nous avons gagné un facteur deux (en temps d'exécution) par rapport à une version n'ayant pas de phase de pré-interprététion. On ne peut pas alourdir outre mesure cette phase de pré-interprétation sans supprimer des possibilités conversationnelles, ni augmenter trop le nouveau pseudo-code, ce qui fait que la charge de l'interpréteur reste encore lourde. L'intérêt de la phase de pré-interprétation est de permettre de traiter certains éléments absents de la première version telles que classes de mémoire et variables implicites.

De plus, pour les fonctions de conversions à appliquer aux valeurs, il semble que le temps passé pour les calculs de précisions (p,q) soit important (car les tests préalablesn'entraînent pas de gain de temps considérable). Il aurait été intéressant, puisque les valeurs de p et q sont connues lors de la pré-interprétation, de conserver comme pseudo-code le code machine nécessaire à leurs calculs, ou même de les calculer.

Une solution pour améliorer les performances de tels compilateurs sans en enlever les possibilités d'interaction est d'effectuer une optimisation au niveau de l'incrément lui-même. Une fois que l'on a interprété un incrément, on peut connaître les instruction "machines" nécessaires pour réaliser son exécution. On pourrait les conserver comme pseudo-code et les utiliser lors d'une activation ultérieure.

Le compilateur interpréteur semble un excellent outil pédagogique. Il permet aux étudiants de connaître rapidement la syntaxe de chaque instruction du langage : une instruction n'est acceptée par le générateur que si elle est correcte "localement" du point de vue syntaxique. Dans la conception des différents messages d'erreurs, il est possible d'en prévoir un certain nombre forçant l'étudiant à plus de rigueur dans la conception de ses programmes. Par exemple, on peut lui demander de déclarer explicitement les variables ou, tout au moins, le prévenir (par un message n'interrompant pas l'exécution) des types pris pour les variables non déclarées. Pour un langage comme PL/1 où beaucoup de conversions sont possibles, de tels garde-fous permettraient à l'étudiant débutant d'éviter, lors de l'exécution, des erreurs qui sont souvent difficiles à déceler.

Les compilateurs interprétatifs peuvent avoir un avenir dans l'apprentissage du langage de programmation et aussi dans la mise au point de programmes du fait des possibilités d'interaction permises à l'exécution.

Du fait de leur "inefficacité" les compilateurs interpréteurs ne sont pas compétitifs avec les compilateurs "batch", et l'on ne peut pas utiliser ces compilateurs pour une utilisation fréquente. Il faut donc que l'utilisation d'un compilateur interpréteur soit compatible avec l'utilisation d'un compilateur "batch".

BIBLIOGRAPHIE

[BAJAR] V. BAJAR, D. CLAUZEL
Le compilateur FORTRAN IV conversationnel
(Notes internes IMAG
Novembre 1970)

[BELLINO] J. BELLINO, Ph. POTIN
LCMS (sous système conversationnel fonctionnant sous OS/360)
Etude du Centre Scientifique IBM (n° FF0106).

[BERGE] C. BERGE
Graphes et hypergraphes
(Ed. Dunod).

[BERT.GRIFF] M. BERTHAUD, M. GRIFFITHS
Incremental compilation and conversationnal interpretation
Annual Review of Automatic Programming
(Vol.7 - Part 2, 1973).

[BRETA] B. BRETAGNOLLE
Compilateur incrémental d'Algol 60
(Notes Internes 1972-73).

[CPL 1] B. LORHO, M. VATOUX
CPL 1 - 1ère Partie - Analyseur générateur
Sept. 1971 - cahier n° 6.

C. BLAIZOT, L. BLAIZOT
CPL 1 - 2ème Partie - Interpréteur conversationnel
Mai 1972 - Cahier IRIA n° 10.

[CUN] Y. CUNIN, PILARD
Décompilateur algol incrémental
1969-70 - Projet de 3ème année ENSIMAG.

[GRIFFITHS 1] M. GRIFFITHS, M. PECCOUD, M. PELTIER
Incrémental Interactive compilation
(IFIP Edinbourg 1968).

[GRIFFITHS 2] M. GRIFFITHS
Cours de compilation
Université de Grenoble.

[GRIFFITHS 3] M. GRIFFITHS
Compiler Construction
Springer Verlag - Berlin Heidelberg - New-York (1974).

[GRIFF-PELT] M. GRIFFITHS, M. PELTIER
A macro-generable language for the 360 computers
(Computer . bulletin - Vol 13 n° 11, novembre 1969).

[PAIR] C. PAIR
Cours de compilation
Ecole d'Eté d'Informatique (Neuchâtel, septembre 1972).

[RAMAMOORTHY] C.V. RAMAMOORTHY
Analysis of graphs by connectivity considerations
(Journal of A.C.M., Vol. 13, n° 2, April 1966, p211-222).

[WARSHALL] S. WARSHALL
A theorem on boolean matrices
(Journal of A.C.M., Vol. 9, 11-12, Janvier 1962).