Mise au point interactive de programmes dans un atelier de génie logiciel

Christian Lenne

To cite this version:

Christian Lenne. Mise au point interactive de programmes dans un atelier de génie logiciel. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1985. Français. NNT : . tel-00315957

HAL Id: tel-00315957
https://theses.hal.science/tel-00315957

Submitted on 2 Sep 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée à

l'Université Scientifique et Médicale de Grenoble

pour obtenir le grade de

DOCTEUR DE 3ème CYCLE

«Informatique»

par

Christian LENNE

MISE AU POINT INTERACTIVE DE PROGRAMMES

DANS UN ATELIER DE GENIE LOGICIEL.

Thèse soutenue le 30 mai 1985 devant la commission d'examen.

S. KRAKOWIAK	Président
E. ANDRE	
P. GREUSSAY	Examinateurs
J. MOSSIERE	
M. SCHLUMBERGER	

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : M. TANCHE

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

(RANG A)

SAUF ENSEIGNANTS EN MEDECINE ET PHARMACIE

PROFESSEURS DE 1ère CLASSE

ARNAUD Paul	Chimie organique
ARVIEU Robert	Physique nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S.
AYANT Yves	Physique approfondie
BARBIER Marie-Jeanne	Electrochimie
BARBIER Jean-Claude	Physique expérimentale C.N.R.S. (labo de magnétisme)
BARJON Robert	Physique nucléaire I.S.N.
BARNOUD Fernand	Biosynthèse de la cellulose-Biologie
BARRA Jean-René	Statistiques - Mathématiques appliquées
BELORISKY Elie	Physique
BENZAKEN Claude (M.)	Mathématiques pures
BERNARD Alain	Mathématiques pures
BERTRANDIAS Françoise	Mathématiques pures
BERTRANDIAS Jean-Paul	Mathématiques pures
BILLET Jean	Géographie
BONNIER Jean-Marie	Chimie générale
BOUCHEZ Robert	Physique nucléaire I.S.N.
BRAVARD Yves	Géographie
CARLIER Georges	Biologie végétale
CAUQUIS Georges	Chimie organique
CHIBON Pierre	Biologie animale
COLIN DE VERDIERE Yves	Mathématiques pures
CRABBE Pierre (détaché)	C.E.R.M.O.
CYROT Michel	Physique du solide
DAUMAS Max	Géographie
DEBELMAS Jacques	Géologie générale
DEGRANGE Charles	Zoologie
DELOBEL Claude (M.)	M.I.A.G. Mathématiques appliquées
DEPORTES Charles	Chimie minérale
DESRE Pierre	Electrochimie
DOLIQUE Jean-Michel	Physique des plasmas
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques pures
GAGNAIRE Didier	Chimie physique

.../...

GASTINEL Noël	Analyse numérique - Mathématiques appliquées
GERBER Robert	Mathématiques pures
GERMAIN Jean-Pierre	Mécanique
GIRAUD Pierre	Géologie
IDELMAN Simon	Physiologie animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques pures
JULLIEN Pierre	Mathématiques appliquées
KAHANE André (détaché DAFCO)	Physique
KAHANE Josette	Physique
KOSZUL Jean-Louis	Mathématiques pures
KRAKOWIAK Sacha	Mathématiques appliquées
KUPTA Yvon	Mathématiques pures
LACAZE Albert	Thermodynamique
LAJZEROWICZ Jeannine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre	Mathématiques appliquées
DE LEIRIS Joël	Biologie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences nucléaires I.S.N.
LOUP Jean	Géographie
MACHE Régis	Physiologie végétale
MAYNARD Roger	Physique du solide
MICHEL Robert	Minéralogie et pétrographie (géologie)
MOZIERES Philippe	Spectrométrie - Physique
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (biologie végétale)
PAYAN Jean-Jacques (détaché)	Mathématiques pures
PEBAY PEYROULA Jean-Claude	Physique
PERRIAUX Jacques	Géologie
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
RASSAT André	Chimie systématique
RENARD Michel	Thermodynamique
RICHARD Lucien	Biologie végétale
RINAUDO Marguerite	Chimie CERMAV
SENGEL Philippe	Biologie animale
SERGERAERT Francis	Mathématiques pures
SOUTIF Michel	Physique
VAILLANT François	Zoologie
VALENTIN Jacques	Physique nucléaire I.S.N.
VAN CUTSEN Bernard	Mathématiques appliquées
VAUQUOIS Bernard	Mathématiques appliquées
VIALON Pierre	Géologie

PROFESSEURS DE 2ème CLASSE

ADIBA Michel	Mathématiques pures
ARMAND Gilbert	Géographie

.../...

AURIAULT Jean-Louis	Mécanique
BEGUIN Claude (M.)	Chimie organique
BOEHLER Jean-Paul	Mécanique
BOITET Christian	Mathématiques appliquées
BORNAREL Jean	Physique
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CHARDON Michel	Géographie
COHENADDAD Jean-Pierre	Physique
DENEUVILLE Alain	Physique
DEPASSEL Roger	Mécanique des fluides
DOUCE Roland	Physiologie végétale
DUFRESNOY Alain	Mathématiques pures
GASPARD François	Physique
GAUTRON René	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude (M.)	Sciences nucléaires I.S.N.
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques appliquées
HERBIN Jacky	Géographie
HICTER Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKOVE Claude (M.)	Géologie
LE BRETON Alain	Mathématiques appliquées
LONGEQUEUE Nicole	Sciences nucléaires I.S.N.
LUCAS Robert	Physiques
LUNA Domingo	Mathématiques pures
MASCLE Georges	Géologie
NEMOZ Alain	Thermodynamique (CNRS - CRTBT)
OUDET Bruno	Mathématiques appliquées
PELMONT Jean	Biochimie
PERRIN Claude (M.)	Sciences nucléaires I.S.N.
PFISTER Jean-Claude (détaché)	Physique du solide
PIBOULE Michel	Géologie
PIERRE Jean-Louis	Chimie organique
RAYNAUD Hervé	Mathématiques appliquées
ROBERT Gilles	Mathématiques pures
ROBERT Jean-Bernard	Chimie physique
ROSSI André	Physiologie végétale
SAKAROVITCH Michel	Mathématiques appliquées
SARROT REYNAUD Jean	Géologie
SAXOD Raymond	Biologie animale
SOUTIF Jeanne	Physique
SCHOOL Pierre-Claude	Mathématiques appliquées
STUTZ Pierre	Mécanique
SUBRA Robert	Chimie
VIDAL Michel	Chimie organique
VIVIAN Robert	Géographie

à Monique

Je tiens à remercier :

Mr S. Krakowiak, Professeur à l'Université de Grenoble, de m'avoir accueilli dans son équipe et de me faire l'honneur de présider le jury de cette thèse.

Mr E. André, Responsable du projet Concerto, pour tout l'intérêt qu'il a porté à ce travail et qui a grandement contribué à la réalisation matérielle de ce projet.

Mr P. Greussay, Professeur à l'Université de Paris 7, qui a accepté de juger ce travail.

Mr J. Mossière, Directeur du Laboratoire de Génie Informatique, qui a dirigé cette thèse. Qu'il me soit permis de lui témoigner ici toute ma reconnaissance pour les précieux conseils qu'il m'a donné lors de la rédaction de ce document.

Mr M. Schlumberger, Responsable du centre de recherche de CAP-SOGETI INNOVATION, d'avoir bien voulu faire partie de ce jury.

Mr M. Santana, mon camarade de travail, d'avoir corrigé la première version de cette thèse et qui par ses conseils et ses critiques m'a apporté une aide inappréciable.

Mrs J-C. Hochain et J. Paris, et tous les membres de l'équipe Adèle avec qui j'ai pu travailler et échanger des idées.

Mrs D. Iglésias et C. Anguille, et le Service de Reprographie de l'Institut IMAG, pour l'excellente qualité de leur travail.

Enfin Nicole et Jean-Louis, qui dans des moments difficiles m'ont permis de reprendre mes études.

Christian Lenne.

Ce travail a été financé par le CNET Lannion dans le cadre du projet CONCERTO.

TABLE DES MATIERES

-o-o-o-

I. INTRODUCTION 1

II. LA MISE AU POINT : PROBLEMES ET SOLUTIONS 5

1. Présentation du problème 5

2. Solutions à la mise au point 8

3. Metteur au point adaptés aux systèmes 9
3.1 Le metteur au point du PERQ 10
3.2 Le metteur au point d'UNIX : ADB 12
3.2.1 Format A.OUT 12
3.2.2 Fonctionnalités de ADB 14
3.3 Le metteur au point de MULTICS : PROBE 20
3.3.1 Structure du code objet MULTICS 20
3.3.2 Commandes PROBE 24

4. Mise au point dans les environnements de programmation ... 28
4.1 PDE1L 30
4.1.1 Interface utilisateur 31
4.1.2 Editeur 32
4.1.3 Commandes de mise au point 33
4.2 INTERLISP 35
4.2.1 Fonction de trace 36
4.2.2 Arrêts conditionnels 37
4.2.3 Retour arrière 37
4.3 DICE 38
4.3.1 Trace de variables 41
4.3.2 Pas à pas 42
4.3.3 Points d'arrêt conditionnels 42
4.4 CPDS 43
4.4.1 Commandes de mise au point 46

5. Conclusion 48

III. UNE SOLUTION A LA MISE AU POINT 51

1. Cadre de travail 51
1.1 Le système d'archivage 53
1.2 L'introducteur 53
1.3 Le générateur de code 53
1.4 L'éditeur syntaxique 54
1.5 L'interface utilisateur 54
1.6 Présentation des fonctionnalités du metteur au point . 54

2. La représentation interne ADELE 57

3. L'interpréteur 60
3.1 Structure des programmes PASCAL 60
3.2 Allocation de variables 62
3.3 Gestion de la mémoire 63
3.4 Traitement des types 64
3.4.1 Les types de base 64
3.4.2 Les types structurés 65
3.4.3 Le type pointeur 69
3.5 Gestion dynamique de la mémoire 70
3.5.1 Gestion de la pile 70
3.5.2 Gestion du tas 71
3.5.3 Ramasse-miettes 72
3.6 Procédures et fonctions prédéfinies 73
3.6.1 Manipulation de fichiers 74
3.6.2 Procédures de gestion dynamique de la mémoire .. 74
3.6.3 Procédures de transfert de données 75
3.6.4 Fonctions diverses 75
3.7 Noyau de l'interpréteur 76
3.8 Evaluateur d'expressions 80
3.8.1 Algorithme de l'évaluateur 82
3.9 Traitements des instructions 83
3.9.1 Instruction d'affectation 83
3.9.2 Instruction BEGIN 84
3.9.3 Instruction IF 85
3.9.4 Instruction CASE 86
3.9.5 Instruction WHILE 87
3.9.6 Instruction REPEAT 87
3.9.7 Instruction FOR 88
3.9.8 Instruction CALL 89
3.9.9 Instruction GOTO 90
3.9.10 Instruction WITH 91

4. Le metteur au point 93
4.1 Fonctionnalités 93
4.1.1 Modes d'exécution 93
4.1.2 Visualisation du programme 97
4.1.3 Manipulation de variables 100
4.1.4 Manipulation de points d'arrêt 106
4.1.5 Accès aux informations liées à l'exécution 108
4.2 Contextes d'exécution 112
4.3 Exemple d'une session de mise au point 113
4.3.1 Programme à mettre au point 114
4.3.2 Session de mise au point 115

5. Aspects interactifs du metteur au point 119
5.1 Dialogue Metteur au point utilisateur 119
5.1.1 Informations à visualiser 119
5.1.2 Informations à désigner 122
5.2 Expériences réalisées 123
5.2.1 Expérience sur un écran alphanumérique 123
5.2.2 Expérience sur deux écrans alphanumériques 124
5.2.3 Expérience sur un écran bit-map 125

IV. EVALUATION DU SYSTEME 127

1. Etat des développement 127
   1.1 Le metteur au point de l'atelier ADELE 127
   1.2 Le metteur au point de l'environnement CONCERTO 129

2. Evolutions souhaitées 131

3. Perspectives 133

ANNEXE A.1 : Fonctions de mise au point.

ANNEXE A.2 : Syntaxe de la RI ADELE.

ANNEXE A.3 : Interface RI mémoire : procédures.

BIBLIOGRAPHIE

I. INTRODUCTION

Dans la dernière décennie, les méthodes de travail en informatique ont profondément été bouleversées par l'apparition en masse de terminaux évolués. Ces nouveaux moyens ont largement favorisé le travail interactif au détriment du mode train de travaux ; en effet, celui-ci n'est plus utilisé que pour les longues exécutions de programmes ne nécessitant pas l'interactivité.

En même temps, côté logiciel de nombreux langages de programmation de haut niveau ont fait leur apparition. La programmation d'applications a donc bénéficié de ces progrès, mais l'absence d'outil de mise au point s'est faite alors sentir. Il a donc été nécessaire d'offrir aux programmeurs un nouveau type d'outil, le metteur au point, permettant de faciliter cette phase du développement. Des études statistiques ont montré que la mise au point interactive permet de localiser les erreurs de programmation dans un temps inférieur de 50 à 300 % par rapport à la mise au point "batch", le temps d'utilisation du calculateur n'augmentant alors que de 30 % (<Sackman 68>).

Ces outils ont ensuite évolué avec les langages de programmation, de plus en plus puissants. En étudiant les différents metteurs au point des systèmes actuels, on constate que si les problèmes posés par la mise au point de programmes sont maitrisés, les solutions à celle-ci n'ont pas toujours suivi la révolution technologique des quinze dernières années. En effet, les systèmes d'affichage offrent à l'heure actuelle de nouvelles possibilités, tant sur le plan de la désignation que sur le plan de l'affichage. Les terminaux alphanumériques font place petit à petit aux écrans à point appelés écrans "bit-map". Depuis quatre ans environ, des dispositifs de désignation directe du type souris sont offerts sur un grand nombre de petits systèmes. Ces micros-ordinateurs ont une puissance de calcul locale de plus en plus importante entièrement

disponible pour l'application. L'outil de mise au point de programmes peut donc bénéficier de toutes ces capacités et doit également mettre en évidence sur les supports de visualisation un grand nombre d'informations contenues dans le programme exécutable.

On constate que les derniers systèmes de mise au point diffèrent peu sur le plan des fonctionnalités des metteurs au point datant d'une quinzaine d'années. La différence porte essentiellement sur la quantité d'informations offerte au programmeur et pour les derniers débogueurs, sur l'aspect communication homme-machine. Dans ce dernier cas, il est parfois difficile de déterminer la frontière entre le metteur au point et l'interface usager.

Cette thèse présente différents types de metteurs au point, puis elle essaye d'apporter une solution à ce problème en présentant le metteur au point PASCAL réalisé au sein de l'environnement de programmation ADELE. Pour cela, elle décrit l'interpréteur réalisé dans le cadre de ce projet ; cet interpréteur travaille sur une représentation interne de l'arbre abstrait des programmes. Nous montrons ensuite comment est exploité cet outil pour réaliser les différentes fonctions de mise au point offertes à l'utilisateur.

Plan de la thèse.

Ce document présente au chapitre 2 les différents problèmes posés par la mise au point de programmes. Il met d'abord en évidence les différentes informations nécessaires à l'utilisateur pour que celui-ci puisse rapidement avancer dans cette phase. Il détaille ensuite les fonctionnalités des systèmes actuels. Comme nous le verrons ces systèmes offrent pratiquement tous les mêmes fonctions, mais la façon de les implémenter varie suivant le support de code (arbre ou code objet). Enfin, il met en évidence, les différentes solutions adoptées sur le plan interface usager.

Le chapitre 3 présente le cadre de travail dans lequel a été effectuée cette thèse, puis décrit les réalisations. Il présente

d'abord l'interpréteur que nous avons réalisé, puis détaille ses fonctionnalités de mise au point. Ce chapitre se termine par une brève description de l'interface usager implémentée.

Le chapitre 4 fait le point sur le système que nous avons réalisé et essaye d'évaluer les prototypes construits. Enfin, la conclusion présente les différentes améliorations à réaliser ainsi que les adjonctions à faire à ce système.

Les annexes sont ensuite constituées de la définition de la représentation interne (R.I) sur laquelle nous avons travaillé et de la syntaxe réelle des commandes de mise au point.

CHAPITRE 2

La mise au point : problèmes et solutions

La mise au point des programmes est une phase importante dans le cycle de vie du logiciel car c'est la plus longue et la plus délicate. Il est donc intéressant d'essayer de gagner en temps, mais aussi en fiabilité sur cette étape. L'objectif de la mise au point n'est pas de vérifier la conformité des spécifications d'un logiciel par rapport à leur mise en oeuvre, mais d'éliminer le plus grand nombre possible d'erreurs de programmation commises lors de cette réalisation. Pour cela, il est intéressant d'avoir un outil permettant de suivre l'exécution du programme, de vérifier la valeur des variables, etc. Cet outil est ce que l'on appelle un metteur au point.

1. PRESENTATION DU PROBLEME

Au cours du développement d'un programme, il apparait généralement dans un premier temps des erreurs simples à localiser et à corriger ; cependant des erreurs plus difficilement détectables persistent et celles-ci sont provoquées par la combinaison d'erreurs antérieures.

En l'absence de metteur au point, le programmeur possède uniquement deux informations pour localiser une erreur. La première, fournie par le système, lui indique l'adresse de l'instruction où l'erreur à été détectée ainsi que le type d'erreur ; dans certains cas, l'adresse est remplacée par le numéro de ligne source correspondant à cette instruction. Ces informations sont liées à la détection par le système d'une erreur d'exécution et ne sont en général que très peu utiles ; en effet, l'erreur n'est dans la plupart des cas qu'un effet secondaire de la véritable erreur (l'erreur de programmation) et, de plus, elle se produit généralement beaucoup plus tard et à un

endroit qui peut n'avoir aucun rapport avec la cause de celle-ci.
La deuxième classe d'information est fournie par le programme lui-même et est constituée des impressions que celui-ci effectue. Ces impressions constituent une trace de l'exécution du programme et peuvent permettre au programmeur de se donner une idée du déroulement de celui-ci ; elles sont bien souvent suffisantes pour trouver certaines erreurs de programmation (initialisations, conditions d'arrêts, etc). mais deviennent rapidement insuffisantes dès que l'on commence à rencontrer des erreurs liées à l'algorithme et aux relations entre les différents composants du programme. Pour détecter ces erreurs, le programmeur rajoute alors à son code des instructions permettant de suivre de plus près le flot d'exécution de son programme : impression du nom de la procédure appelée, région du programme en cours d'exécution, impression des valeurs de certaines variables d'état du programme, etc.
Avec toutes ces informations, le programmeur essaie alors de remonter dans son programme, à partir de l'erreur détectée par le système, en essayant de mettre en évidence la cause de celle-ci. Pour cela, il simule sur papier le déroulement qu'a suivi son programme en notant les valeurs successives de chaque variable ; il cherche ensuite à calculer ou à déduire la valeur des autres variables. Cette méthode est très laborieuse et peu fiable car elle demande beaucoup de concentration et oblige le programmeur à réaliser tous les calculs intermédiaires, même si ceux-ci n'apportent aucune information importante.

Les metteurs au point ont donc pour objectif l'automatisation de ces différentes tâches.
Avec les anciens systèmes, on disposait de fonctions rudimentaires pour faciliter la mise au point de programmes. De tels systèmes offrent d'une part des appels à des fonctions de trace que l'utilisateur doit placer dans son programme là où il le juge utile. Ces fonctions permettent de suivre le déroulement partiel de l'exécution et en même temps de visualiser le contenu de certaines variables. Cette méthode comporte différents inconvénients majeurs. En effet, il est nécessaire d'appréhender

le comportement de son programme pour placer judicieusement les ordres de trace. Malheureusement, le programme ne passe pas systématiquement dans les branches prévues, surtout lorsque l'imbrication des instructions de test rendent fortement combinatoire les chemins d'exécution du programme. De ce fait, il est nécessaire de remodifier le progamme source pour rajouter de nouveaux tests, ce qui implique une recompilation du programme. Il faut également déterminer les variables significatives dont le contenu est à tracer.

D'autre part, lorsque le programme perd le contrôle, le système édite une image mémoire (Post-Mortem Dump). Cette liste donne (en hexadécimal ou en octal) l'état, au monment de l'erreur de toute la partition de mémoire allouée au programme (instructions et données). On peut grâce à cette liste, analyser la valeur des variables du programme à cet instant.

Toutefois, cet état donne une masse d'informations peu significatives, car dans la plupart des cas, l'erreur de programmation est loin de l'erreur d'exécution, et l'état de la mémoire ne permet pas de remonter à la cause de celle-ci.

Cependant, cette façon de procéder était relativement bien adaptée aux systèmes de l'époque, c'est à dire au mode train de travaux.

L'évolution technologique a changé profondément les méthodes de travail ; la réalisation de programme s'effectue interactivement sur des terminaux à écran. Les metteurs au point actuels ont donc exploité l'interactivité offerte par les systèmes pour suivre le déroulement des programmes. Il n'est ainsi plus nécessaire de recompiler son programme pour insérer des instructions de mise au point. Ces fonctions de mise au point permettent au programmeur de suivre et de contrôler le déroulement de son programme de façon interactive. Il peut ainsi juger en temps réel de la nécessité de poser un point d'arrêt ou de visualiser certaines variables. Il peut également corriger certaines valeurs pour pouvoir poursuivre l'exécution de son programme.

La mise au point interactive nécessite également une interface de visualisation adaptée à l'utilisateur. Il est par exemple possible de visualiser les variables de différentes façons, c'est

à dire, éditer leurs valeurs en hexadécimal ou en décimal en ignorant leurs types de données, mais également de visualiser leurs valeurs en fonction de leurs types. Des études ponctuelles ont porté sur la visualisation au cours de la mise au point du contenu de variables de type structurées (<Myers 80>). Les algorithmes mis en oeuvre (en particulier pour les variables de type record PASCAL) sont très complexes et notre étude n'abordera pas ce point précis.

La désignation d'une variable peut, en outre, se faire soit en donnant son adresse et sa longueur, soit en donnant uniquement son nom. Dans le premier cas, il est nécessaire de connaître l'implantation de la variable ainsi que la manière dont le compilateur traite les différents types de données ; la première de ces informations est généralement donnée sur la liste produite par le compilateur mais la seconde nécessite une certaine connaissance du compilateur. Dans le deuxième cas, le système est capable par lui-même de retrouver l'adresse et le type de la variable.

2. SOLUTIONS A LA MISE AU POINT

La mise au point de programme peut se faire de deux façons différentes. La première, la plus courante, consiste à intégrer au code objet durant la phase de compilation un certain nombre d'informations (tables, déroutements, indicateurs, ...). Ces extensions sont générées sur demande lors de la compilation du module. Cette méthode permet généralement de visualiser le code source, d'accéder aux variables, etc., et de conserver l'efficacité d'exécution des programmes compilés. Nous décrivons au paragraphe 3, trois metteurs au point fonctionnant sur le code objet. Le premier système est le metteur au point intégré au système POS du PERQ (<ICL 82>). Le second système est le metteur au point ADB, utilitaire du système UNIX (<Unix 79>). Le troisième est le metteur au point PROBE (<Probe 83>) du système MULTICS.

L'autre solution permettant la mise au point de programme est la réalisation d'un interpréteur adapté au langage de

programmation. Un analyseur construit alors une représentation intermédiaire du programme à exécuter, représentation sur laquelle travaillera l'interpréteur. Une fois ce programme au point, le texte source pourra être compilé pour accélérer l'exécution.

Les interpréteurs sont dans ce deuxième cas des outils, au même titre que les compilateurs, éditeurs, etc. Ils peuvent également être jumelés à un éditeur. C'est le cas d'un bon nombre d'interpréteurs s'exécutant dans les environnements de programmation. Dans ce dernier cas, ces outils ne peuvent travailler qu'au sein de cet environnement.

Nous présentons au paragraphe 4 quelques outils de mise au point de la deuxième famille.

3. METTEUR AU POINT ADAPTES AUX SYSTEMES

Situons tout d'abord la phase de mise au point dans le cycle d'écriture d'un logiciel. Le premier schéma illustre la chaine d'écriture classique d'un programme :

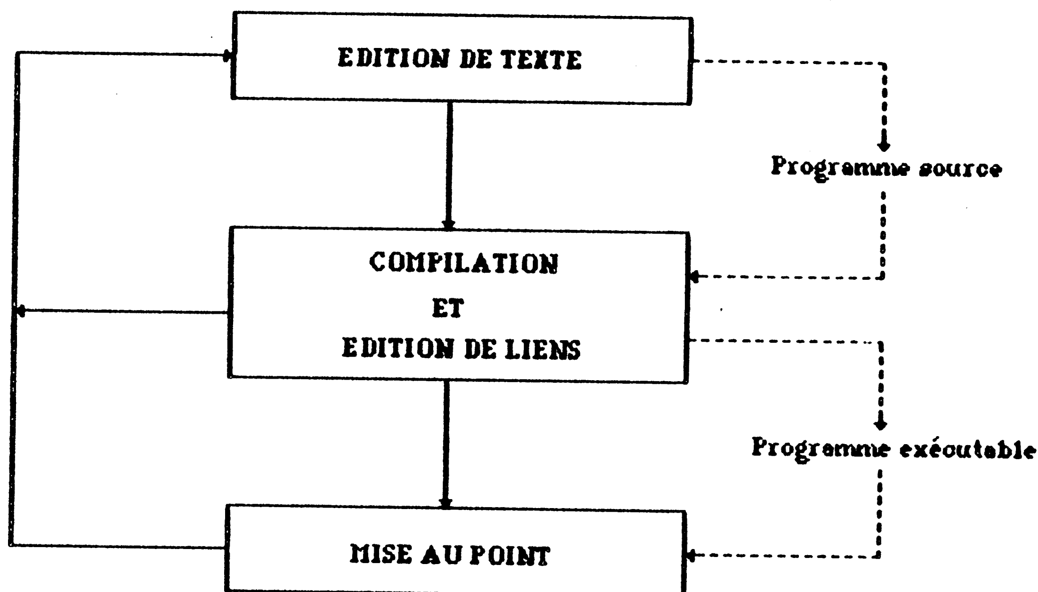


FIGURE 1

Nous avons regroupé les phases de compilation et d'édition de liens dans une seule phase. Ceci n'est pas général à tous les systèmes, car certains systèmes actuels offrent une édition de

liens dynamique.

Dans ce cycle d'écriture la mise au point se fait directement sur le code objet. Nous allons maintenant décrire un premier type de metteur au point développé sur le PERQ.

3.1. Le Metteur au point du PERQ

Le PERQ est un mini-ordinateur (ordinateur personnel) développé à l'université de Carnegie Mellon et commercialisé en Europe par ICL. Ses caractéristiques essentielles sont :

- un écran graphique à haute résolution adressable par souris,
- un processeur 16 bits en tranches de 4 bits.

Le système d'exploitation implanté sur cette machine, POS, offre à l'utilisateur une interface très agréable, exploitant au maximum les possibilités graphiques du calculateur. L'utilisateur est guidé par un menu affichable à la demande, et accessible par souris. Un grand nombre de procédures de gestion d'écran sont à la disposition de l'utilisateur et permettent de gérer dynamiquement un ensemble de fenêtres. Ce système très puissant comporte en outre un éditeur plein écran d'usage très agréable et facilement assimilable.

Le langage de programmation est unique. C'est un PASCAL étendu qui autorise la modularité, la manipulation de chaines de caractères, etc. Le compilateur génère un P-Code interprété par le microcode de la machine. Le code produit comporte très peu d'extensions, et nous allons voir qu'un tel système n'offre à l'utilisateur qu'un service de mise au point de bas niveau.

L'ensemble des fonctions de mise au point est très restreint. Il existe peu de liens entre le programme source et le programme objet. Les seuls liens existants sont des références entre l'appel d'une procédure et son nom ; de plus à chaque instruction est attaché un numéro correspondant à la ligne source. Il convient donc de toujours avoir une liste de compilation à jour car il n'existe aucune fonction interactive pour lister la ligne ayant

provoqué l'erreur. Le scénario classique d'une session de mise au point sur ce système est le suivant.

RUN Programme_exe

L'exécution du programme est alors lancée. Lorsqu'une erreur d'exécution apparait, celle-ci est signalée à l'utilisateur. L'appel de la fonction mise au point se fait alors explicitement. Cette commande a pour action d'éditer le contenu de la pile d'exécution. La visualisation de cette pile met en évidence les informations suivantes :

- Le numéro de ligne ayant provoqué l'appel d'une procédure,
- Le nom de la procédure appelante,
- Le nom du programme dans lequel se trouve la procédure ayant provoqué l'appel.

Le nom de la procédure figurant dans la pile est tiré du fichier source et non du fichier objet ; toutefois aucune vérification n'est faite sur la cohérence entre ces deux fichiers. Si le fichier source n'existe pas, seul le numéro de la procédure est édité.

Les fonctions de mise au point disponibles sont essentiellement des fonctions de manipulation de la pile. Ces fonctions permettent de se déplacer dans celle-ci. Leur but est d'accéder à l'environ de la procédure correspondant au niveau désigné. Une commande permet de connaitre la taille de l'environ local, et une autre celle de l'environ global. La désignation d'une variable ou d'un paramètre se fait en donnant le déplacement relatif au début de l'environ, information qui figure sur la liste de compilation. Le mode d'édition de la valeur de la variable peut être sélectionné par une commande ; il est alors possible d'imprimer la valeur associée à une adresse suivant le format correspondant à son type. Les types traités sont : entier, caractère, booléen, chaine. Pour ce dernier, la longueur de la chaine est également affichée.

Conclusion

Un tel metteur au point impose d'avoir en permanence une liste de compilation à jour et de suivre l'exécution à l'aide de cette liste.
Ce système de mise au point est donc très primitif : aucune fonction de trace n'est fournie, l'accès aux variables se fait par l'adresse, et il est impossible de mettre des points d'arrêt. Un programme ne provoquant pas d'erreur d'exécution, mais donnant des résultats erronés est donc très difficile à mettre au point avec un tel système. De plus les possibilités graphiques et les fonctions de gestion d'écran offertes par le système, n'ont pas du tout été exploitées pour encapsuler ces fonctions de mise de point.

Le second système que nous avons étudié est également basé sur le code objet et offre à l'utilisateur un jeu de commandes de mise au point beaucoup plus riche que le précédent. Cet outil est le processeur ADB du système UNIX.

3.2. ADB

ADB est le metteur au point du système UNIX permettant de développer des programmes binaires au format A.OUT, format propre à UNIX.
Avant de détailler les différentes commandes de mise au point, nous examinons la structure du format binaire des fichiers de type A.OUT.

3.2.1. Format A.OUT

Les fichiers binaires d'UNIX sont formés de septs parties. Ces différentes parties sont ordonnées de la façon suivante :

a - une en-tête,
b - un segment texte,

c - un segment de données,
d - la table de translation du texte,
e - la table de translation des données,
f - la table des symboles,
g - la table des noms.

a) L'en-tête

Cet en-tête de 32 Octets contient le nombre magique qui indique le type du fichier (binaire, texte, etc.) et l'organisation de certaines sections dans le cas des fichiers binaires. Figurent également dans cette en-tête, la taille des différentes sections et le point d'entrée du programme.

b) Le segment texte

Cette partie contient les instructions qui composent le programme.

c) Le segment données

Ce segment contient les données initialisées du programme.

d et e) Les tables de texte et données relogeables

Ces deux tables contiennent toutes les informations permettant l'implantation en mémoire des instructions et des données.

f) La table des symboles

Cette table est formée d'une suite d'enregistrements ayant le format suivant :

- un double mot indique le symbole concerné en donnant son déplacement dans la table des noms.

- un octet qui indique le type du symbole et sa définition (externe ou non).
- trois octets inutilisés.
- quatre octets contenant l'adresse dans le segment texte.

Dans cette table n'apparaissent que les variables allouées en mémoire. Par conséquent, les variables allouées dans la pile ne sont pas mémorisées.

g) La table des noms

Cette table contient les différents noms référencés dans la table des symboles.

Nous allons voir dans ce qui suit comment sont exploitées ces informations en décrivant les différentes fonctionnalités offertes par ce metteur au point.

3.2.2. Fonctionnalités de ADB

Ce metteur au point a été conçu pour permettre la mise au point de programmes générés par différents compilateurs et par l'assembleur. C'est dans ce dernier cas qu'il est le plus efficace, car il ne lui est pas possible pour un langage évolué de reconstituer l'instruction de base du langage à partir du code objet. La seule fonctionnalité offerte dans ce sens est la génération du source assembleur correspondant (désassemblage). ADB peut travailler sur deux fichiers simultanément qui correspondent au fichier objet à mettre au point, et à l'image mémoire (équivalent du "Dump") d'un programme ayant provoqué une erreur d'exécution. Le contenu de cette image mémoire peut donc être dépouillée à l'aide de certaines commandes de mise au point offertes par ADB, mais aucune ne permet de relancer l'exécution à partir de ce code.

L'ensemble des commandes peut être divisé en quatre classes :

- La première offre la possibilité de visualiser ou de modifier le contenu d'une adresse mémoire.
- La seconde permet de manipuler des variables locales créées sous ADB et indépendantes du programme.
- La troisième contient les différentes commandes permettant d'exécuter le programme.
- Enfin la dernière classe contient l'ensemble des commandes permettant de consulter les informations liées à l'exécution du programme.

a) Visualisation et modification du contenu d'une adresse mémoire

Les commandes permettant d'explorer le contenu de la mémoire ont la forme générale suivante :

<Adresse>,<Compteur> <Séparateur> <Commande>

Elles s'adressent au fichier objet si le séparateur est égal au caractère "?", au fichier image mémoire, si le séparateur est le caractère "/". Pour accéder à une variable du programme, deux cas sont à dissocier. Soit la variable du programme est une variable globale (son nom est mémorisé dans la table des symboles), on accède alors à son adresse par son nom. Le deuxième cas, malheureusement le plus courant, concerne les variables automatiques. Celles-ci sont générées dans la pile et ne sont accessibles que par leur adresse. Cette adresse est alors donnée en hexadécimal, décimal ou octal suivant l'option choisie. Celle-ci peut être également une expression composée du nom d'une variable (variable statique du programme ou locale à ADB), d'une adresse et d'un opérateur ("+", "-", etc.). Parmi ces opérateurs, figure un opérateur d'indirection ("*").

Le paramètre compteur précise le nombre de mots de 16 Bits à afficher. Le champ "Commande" indique le format d'édition à utiliser pour l'affichage de la zone mémoire associée à l'adresse. Si celle-ci fait partie de la zone de code, le programme est

désassemblé. Sinon, suivant la commande donnée la ou les valeurs sont affichées en hexadécimal, décimal, octal ou caractère.

Une commande permet de modifier le contenu d'une adresse. Cette commande doit être suivie des valeurs à stocker à cet endroit. Cette valeur sera chargée dans le mot de 16 Bits désigné.

Par exemple, la commande "_main,2?x" imprime le contenu de cette adresse en hexadécimal, c'est à dire :

```
_main:  2e11 add4
```

Si on veut modifier le contenu du deuxième mot, on donne :
"_main+2?w 1010" ce qui déclenche l'impression :

```
a6:   add4 = 1010
```

Pour désassembler un fragment de programme, on émetttra la commande :
"_main+4,3?i". Celle-ci aura pour effet d'imprimer :

```
a8:      jsr _printf
         addl #12,sp
         unlk a6
```

Toutefois, pour éviter de manipuler exagérément des adresses numériques, ADB offre à l'utilisateur de mémoriser certaines d'entre elles dans des variables locales. L'affectation d'une adresse à une variable locale ADB se fait par la commande : "adresse > nom". La référence à cette variable se fera en précédant le nom de la variable du caractère "<".

Remarque :

Cette fonctionnalité pert un peu de son intérêt car les noms de ces variables sont limités à un caractère alphabétique ou numérique.

Certaines de ces variables sont réservées et mémorisent les principales caractéristiques du code. Ce sont les variables :

b : adresse de base du segment de données.
d : taille du segment de données.
e : adresse du point d'entrée du programme.
m : le nombre magique du programme chargé.
s : la taille du segment pile.
t : la taille du segment texte.

Un autre groupe de commandes permet de lancer et de contrôler l'exécution du code objet chargé sous ADB.

b) Commandes de contrôle d'exécution

Ce groupe de fonctions est constitué de six commandes qui permettent d'une part de lancer ou stopper l'exécution et d'autre part de placer ou supprimer des points d'arrêt. Toutes ces commandes sont préfixées par le caractère ":". Leur structure est la suivante :

"adresse : commande paramètre"

La commande ":r" permet de lancer l'exécution sur le point d'entrée du programme. Un processus est alors créé et celui-ci reste actif jusqu'à ce qu'une condition d'arrêt soit rencontrée. L'exécution peut alors être relancée par la commande "Adresse:c Signal". Si une telle commande est donnée, l'exécution est relancée à l'adresse donnée. Par défaut, cette adresse est le point courant. Le "Signal" qui suit la commande est un signal au sens UNIX. Il est également possible de relancer l'exécution en mode pas à pas par la commande "Adresse:s N" ; dans ce cas, N instructions sont exécutées avant le nouvel arrêt. Si un processus est encore actif et qu'une nouvelle commande ":r" est lancée le processus précédent est détruit implicitement, et un nouveau processus est créé. La commande ":k" permet de tuer explicitement le processus courant.

Pour contrôler l'exécution, il est possible de positionner ou de supprimer dans le fichier objet des points d'arrêt. Au sens ADB, un point d'arrêt est une adresse à laquelle une autre commande ADB sera exécutée. La pose d'un point d'arrêt (commande ":b") est caractérisée par trois paramètres. Le premier est bien entendu l'adresse sur laquelle l'action donnée en troisième paramètre sera exécutée. Le deuxième paramètre précise au système le nombre de fois ou l'exécution de cette instruction ne déclenchera aucune action. On voit donc que cette commande peut être utilisée soit comme point d'arrêt au sens propre du terme, soit comme fonction de trace d'exécution ou d'adresse mémoire.

Un point d'arrêt subsiste jusqu'à ce qu'une commande ":d" le supprime.

Un autre groupe de commandes permettent d'afficher certaines informations contenues dans le code objet et certaines informations liées à l'exécution.

c) Visualisation d'informations liées à l'exécution

Les diffèrentes informations affichables sont :

- la pile d'appels de procédure,
- la table d'adresses,
- les registres de la machine,
- les variables locales à ADB,
- les variables globales du programme,
- les points d'arrêt.

c.1) La pile d'appel

La commande "$c" permet de visualiser la pile d'appel des procédures. Si le programme n'est pas lancé, c'est à dire si aucun processus n'est actif, et si une image mémoire a été chargée, la pile affichée correspond à la pile d'exécution du fichier image mémoire telle qu'elle était au moment de l'erreur. On peut donc,

après une erreur d'exécution, mettre au point son programme et visualiser l'état courant. Cette commande affiche pour chaque procédure active, le nom de la procédure, la valeur des paramètres et l'instruction d'appel.

c.2) La table d'adresses

Cette table indique pour le fichier binaire et pour l'image mémoire l'adresse des différentes sections.

c.3) Les registres de la machine

Les registres de la machine sont visualisés par la commande "$r". Tous les registres sont affichés ainsi que leur contenu, et le compteur ordinal (pc) est désassemblé.

c.4) Les variables locales à ADB

La commande "$v" permet de visualiser la valeur des différentes variables créées sous ADB. Seules les variables différentes de zéro sont affichées.

c.5) Les variables partageables du programme

Les variables partageables peuvent être visualisées sous ADB. La commande "$e" permet de visualiser tous les noms de variables affectées et leur adresse respective.

c.6) Les points d'arrêt

Cette commande permet de visualiser la valeur d'un point d'arrêt, ainsi que la commande à exécuter lors de l'arrêt.

Conclusion

Ce type de metteur au point est très utile pour mettre au point des programmes développés en assembleur. Bien que la possibilité de mise au point de programmes écrits en langage évolué soit offerte, ceci est d'un intérêt moindre car il est nécessaire de connaitre la manière dont le compilateur génère les instructions. De plus, si on veut utiliser un tel outil pour visualiser le contenu des variables, il est nécessaire que le compilateur alloue les variables en dehors de la pile, pour que celles-ci figurent dans la table des symboles.

Le paragraphe suivant présente un autre metteur au point qui exploite les différentes informations contenues dans le code objet. Ce metteur au point est beaucoup plus puissant que ADB, car il tient compte du langage de programmation utilisé pour générer le code objet. C'est le metteur au point PROBE du système MULTICS.

3.3. PROBE

PROBE est un metteur au point applicable à la plupart des langages de programmation disponibles sur Multics ; il est actuellement implémenté pour PL/1, PASCAL, ALGOL68, FORTRAN et COBOL. C'est un metteur au point de haut niveau qui doit sa généralité à la structure du code objet propre au système MULTICS. En effet, PROBE exploite des tables (d'un format général) générées par les différents compilateurs. Ce sont donc ces derniers qui préparent les données nécessaires au metteur au point.

3.3.1. Structure du code objet MULTICS

Un segment de code objet de Multics est composé de six sections. Nous illustrons la structure d'un segment objet dans la figure suivante.

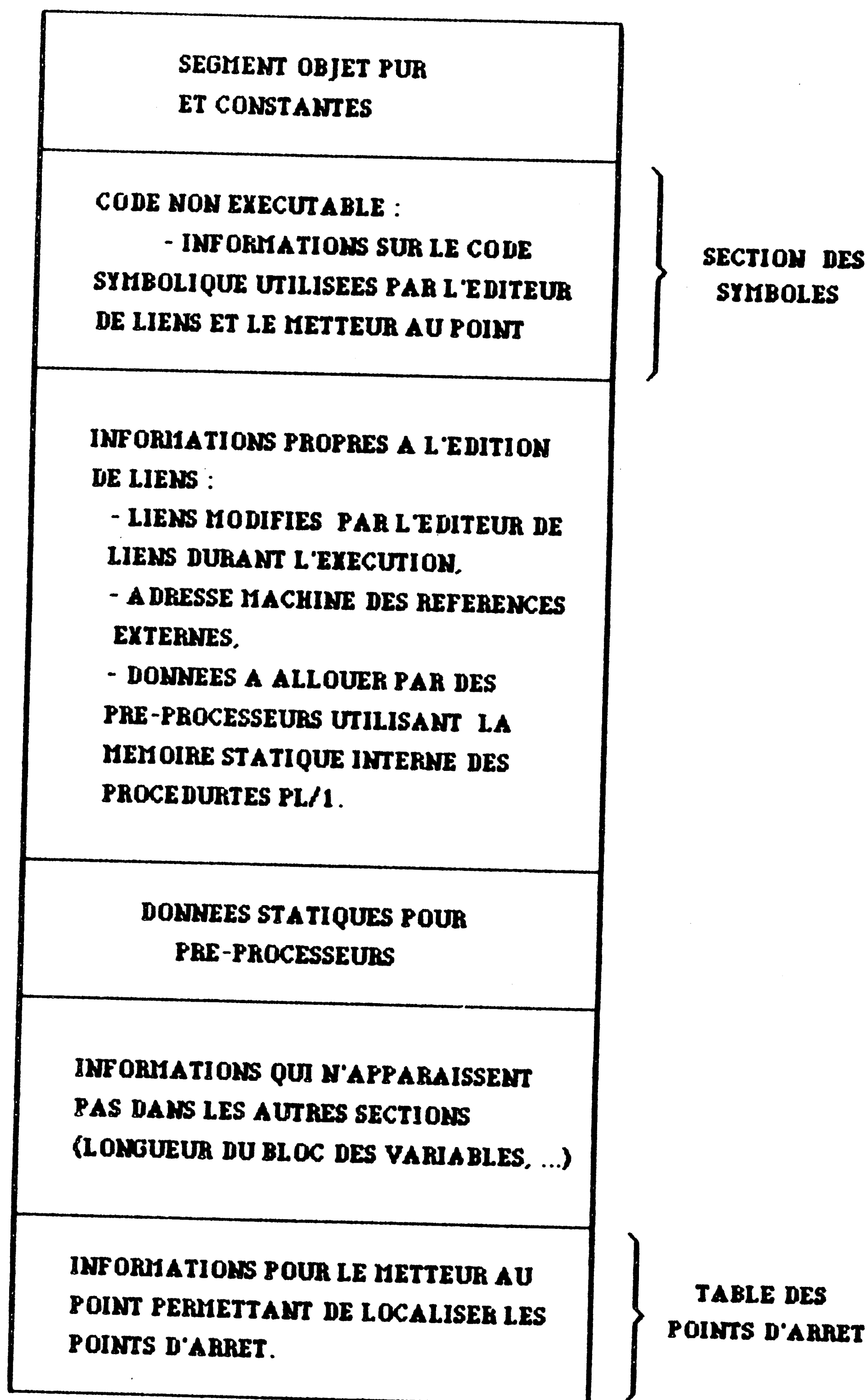


FIGURE 2

Nous ne nous intéressons dans cet exposé qu'à la section des symboles et à la section table de points d'arrêt, car les autres sections sont surtout exploitées par le système et en particulier par l'éditeur de liens dynamiques.

a) Section des symboles

Nous décrivons dans ce paragraphe l'organisation de la table des symboles. Cette table comprend 3 blocs :

- Le bloc des symboles PL/1.
- La table des symboles du noyau d'exécution PL/1.
- La table des instructions.

1) Bloc des symboles PL/1

Ce bloc contient toutes les informations sur la façon dont a été généré le code. Apparaissent dans cette section, des informations diverses telles que :

- Version du compilateur
- Options de compilation
- Nom complet du segment
- etc.

2) Table des symboles

Cette table est construite à la compilation dans un format standard indépendant du compilateur. Elle contient les informations de base sur les symboles déclarés dans le programme. Dans cette table figure également la table des symboles générée par le compilateur. Toutefois, son format est spécifique et la façon dont elle est mémorisée est indépendante du format utilisé par le compilateur.
On y trouve pour chaque variable un ensemble d'informations telles que le type de la donnée, sa classe mémoire, la routine permettant de l'éditer et son adresse qui sera déterminée pendant l'exécution du programme. A chaque identificateur sont associés son nom, sa longueur et un pointeur vers sa déclaration.

De plus, un format d'édition est associé à cette variable (entier, caractère, tableau, etc).
Les niveaux statiques d'un programme sont aussi représentés. A chaque procédure ou début de bloc est associée une entrée dans la table, contenant entre autres des pointeurs sur les symboles définis dans le bloc (label, constante, variable, etc). Cette entrée contient également trois pointeurs permettant d'accéder :

- au point de retour de la procédure ou du bloc,
- aux paramètres de la procédure englobante,
- aux paramètres du bloc.

La visibilité des variables est aussi mémorisée grâce à une hiérarchisation de la structure.

3) La table des instructions

Cette table contient les informations sur les instructions du programme source. A chaque instruction est associé un pointeur vers le code objet associé à celle-ci. Elle est repérée dans le texte source par le numéro de lignes dans le fichier contenant l'instruction, le début de l'instruction dans cette ligne, et sa longueur en nombre de caractères.

b) La table de points d'arrêt

Cette table est gérée par le metteur au point et lui permet de mettre des déroutements à une adresse donnée durant l'exécution.

La structure de ce code objet est beaucoup plus complexe et beaucoup plus riche que celui du code UNIX. Nous allons maintenant voir comment toutes ces informations sont exploitées par PROBE et montrer que cette structure lui

confère une puissance largement supérieure à ADB.

3.3.2. Commandes PROBE

Le langage de commandes de PROBE est très riche : une quarantaine de commandes. Toutefois, certaines sont implémentées uniquement pour un langage ou un groupe de langages et n'ont aucun sens pour les autres. D'autres commandes n'ont, à nos yeux, que très peu d'utilité pour la mise au point : exécution de commandes système, affichage du langage utilisé, affichage du code généré, etc. Nous ne décrirons donc, dans ce chapitre, que les possibilités de PROBE qui nous ont parues intéressantes et que nous avons regroupées en classes liées :

- au mode d'exécution,
- à la manipulation des points d'arrêt,
- à la manipulation de la pile,
- à la manipulation des variables,
- aux commandes diverses.

PROBE gère deux pointeurs au cours de l'exécution d'un programme. Le premier identifie la ligne source de l'instruction courante : c'est le pointeur d'exécution. Le second permet de désigner une ligne du fichier source : c'est le pointeur source. Toute visualisation du texte se fait relativement à ce pointeur. Nous verrons dans la dernière classe de commandes l'exploitation de ces deux pointeurs.

a) Mode d'exécution.

L'exécution d'un programme peut se faire de deux manières distinctes. La première correspond à une exécution classique. Le programme est activé et reste actif jusqu'à ce qu'une condition d'arrêt se produise : fin du programme, erreur d'exécution, point d'arrêt, etc.

L'exécution peut être relancée à une instruction source donnée, à condition que la procédure contenant cette ligne

instruction soit active. Il est également possible d'activer directement une procédure pour la tester : cette procédure peut être interne ou externe, mais elle doit être accessible à partir du point courant. La commande permettant de réaliser cette fonction est du type :

CALL Nom_Procédure <Paramètres>

Le second mode permet d'exécuter le programme pas à pas. Dans ce mode, une instruction est exécutée, et le pointeur d'exécution est placé juste après celle-ci. Le pas d'exécution est une instruction simple c'est à dire qu'une instruction composée, telle qu'un "Begin-End" PASCAL doit être considérée comme la suite de N instructions simples.

b) Manipulation de points d'arrêt.

Il est possible en PROBE de positionner des points d'arrêt avant ou après une instruction. L'exécution du programme sera alors arrêtée à la rencontre de cette instruction, et celle-ci aura ou n'aura pas été exécutée. Cette commande a la syntaxe suivante:

```
AFTER  |
       |-   N° Ligne    ": Commande PROBE"
BEFORE |
```

Par défaut, cette commande prend la ligne courante (désignée par le pointeur source) pour la localisation du point d'arrêt, et la commande "HALT" pour la commande à exécuter lors du passage sur le point d'arrêt. Ces deux commandes sont donc plus que des commandes de pose de point d'arrêt, mais également des commandes de trace. La commande offre également la possibilité de réaliser des points d'arrêts conditionnels.

c) Manipulation de la pile

Seule la visualisation de la pile d'appel est possible. Chaque élément correspondant à une procédure active est constitué des informations suivantes :

- Nom de la procédure,
- Langage d'écriture,
- N° de ligne ou adresse de l'instruction d'appel,
- Type d'erreur si l'exécution de la procédure s'est terminée anormalement.

d) Manipulation de variables

Trois commandes permettent de manipuler les variables du programme ; celles-ci sont désignées par les noms qui leur ont été associés dans le programme. La première commande permet de connaître les attributs associés à une variable ; trois informations sont fournies :

a) la portée de la variable (locale ou globale),
b) le type de la variable,
c) le nom de la procédure de déclaration.

La deuxième commande permet de mettre en évidence le contenu d'une variable. Dans le cas d'un programme PASCAL, l'affichage de la valeur dépend du type de la variable. L'affichage d'une structure Pascal est effectué de la manière suivante :

```
Résultat Commande         Programme source

struct =                  struct : record
  c1 =                      c1 : record
    c11 = 1234                c11 :integer;
    c12 = 'A'                 c12 : char  end ;
  c2 = 'ABCDEFGH'           c2 : packed array [1..10] of char;
  c3 =                      c3 : record
    c31 =                     c31 : record
      c311 [1] = 1              c311 : array [1..10] of
      c311 [2..10] = 0                       integer ;
      c312 = 0                  c312 : integer end ;
    c32 = 'IJKL'              c32 : packed array [1..10] of char end;
  c4 = false                c4 : boolean ;
  c5 = rouge                c5 : (bleu, blanc, rouge)
                          end;
```

Nous voyons dans cet exemple les représentations données aux différents types Pascal. S'il s'agit d'un tableau dont une suite d'éléments ont la même valeur, la valeur de l'indice est remplacée par l'intervalle dans lequel la variable reste inchangée. Les variables de type énuméré sont affichées pour leur part en utilisant leur format de déclaration.

Lorsque des déclarations de variables locales masquent celles d'autres variables globales, seules les locales peuvent être visualisées ; de même, seules les variables du dernier appel d'une procédure appelée récursivement sont visibles.

La troisième commande permet de modifier la valeur d'une variable. La valeur à affecter doit être compatible avec le type de la variable.

e) Commandes diverses

PROBE permet de visualiser les paramètres de la procédure courante. Il permet également de visualiser le programme

source, l'affichage se faisant à partir du pointeur source. C'est l'utilisateur qui indique le nombre d'instructions à afficher. Le pointeur source peut être déplacé sur une autre instruction du programme ; toutefois, si l'exécution est relancée (exécution d'un pas de programme ou exécution en continu), le pointeur source est réinitialisé à la valeur du pointeur d'exécution. A tout moment, il est possible d'afficher la valeur de ces deux pointeurs.

Conclusion

Ce metteur au point multilangage possède donc des commandes très puissantes. Il faut cependant regretter l'absence de visualisation des variables locales à différents niveaux. Nous notons également que l'écriture des procédures permettant l'accès aux variables et la visualisation de ces variables est coûteuse pour les langages typés comme PASCAL. Preuve en est le temps passé pour mettre à niveau le metteur au point PROBE pour le compilateur PASCAL (environ 1 homme/année).

Au travers de l'étude de ces trois metteurs au point, il apparait très nettement que les possibilités offertes sont fortement dépendantes de la structure du code objet. Le côté interface avec l'usager, indépendant de la structure du code, est le plus souvent négligé, même dans le cas du PERQ dont les possibilités graphiques sont pourtant impressionnantes.

4. LA MISE AU POINT DANS LES ENVIRONNEMENTS DE PROGRAMMATION

Les metteurs au point des environnements de programmation sont en général bâtis soit sur des interpréteurs, soit sur des compilateurs incrémentaux. Les techniques d'interprétation offrent d'énormes avantages. Elles permettent en particulier de raccourcir le cycle de développement du logiciel, cycle qui est illustré par la figure suivante.

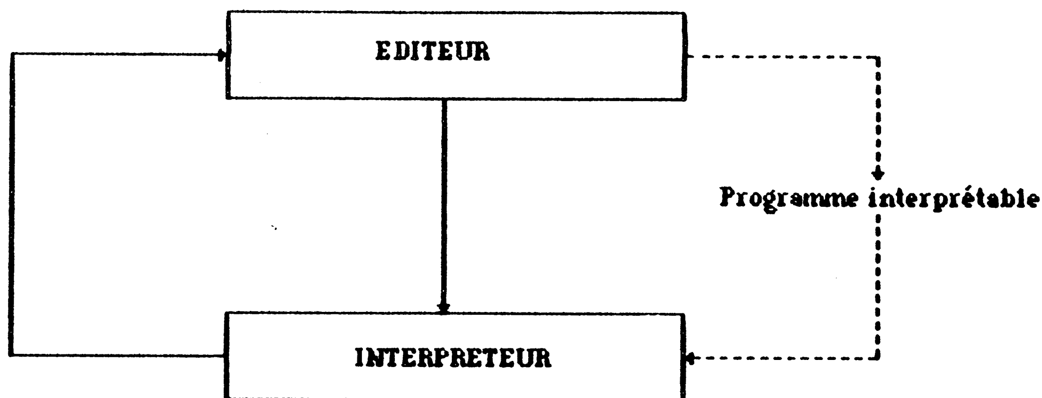


FIGURE 3

A l'heure actuelle, les interpréteurs les plus courant sont ceux des langages de programmation BASIC et LISP.

Les interpréteurs BASIC ont permis une large diffusion de ce langage qui équipe tous les micro-ordinateurs actuels. La facilité de programmation et l'usage de ces systèmes ont été une des clés de réussite de ce langage.

Le fait que ce dernier soit interprété a contribué au succès remporté par ce langage bien que le bas niveau de sa spécification rende souvent difficile la programmation d'une grosse application. Il n'existe par exemple ni la notion de modularité, ni la notion de procédure. L'interprétation immédiate d'instructions permet de limiter très fortement les instructions de mise au point (on trouve uniquement la fonction trace de programme). Par conséquent, l'interprétation immédiate de l'instruction "PRINT variable", provoque l'affichage du contenu de la variable. De la même manière, une affectation entrée directement au clavier modifie le contenu d'une variable. Les commandes de mise au point sont en fait les instructions du langage.

Les seconds interpréteurs sont de plus en plus répandus. Il y a encore quelques années, LISP était utilisé pour la programmation de systèmes d'intelligence artificielle. Aujourd'hui, il est offert sur de nombreux calculateurs (y compris sur les micro-ordinateurs) et utilisé dans divers domaines comme l'intelligence artificielle, le génie logiciel, etc. Cet essor est du bien sûr à la puissance d'expression de LISP, mais aussi au fait qu'il soit un langage interprété.

Nous présentons dans la suite de ce chapitre quatre environnements de programmation différents. Nous ne nous sommes intéressés qu'aux ateliers offrant des possibilités de mise au point ; de ce fait, les systèmes tels que MENTOR (<Douzeau-Gouge 79>) ou GANDALF (<Haberman 82>) ont été volontairement laissés de côté.

Les quatres systèmes étudiés sont :

- PDE1L,
- INTERLISP,
- DICE,
- CPDS.

4.1. PDE1L

PDE1L (Program Development Environment for PL1L (<Mikelsons 80>)) est un environnement de programmation développé au sein d'IBM. Cet environnement a été réalisé après une première expérience acquise lors de la réalisation d'un environnement LISP (<Mikelsons 80a>), dont la philosophie était dans la même lignée que PDE.

PDE1L est un ensemble d'outils d'aide à l'écriture et au test de programmes PL1L (PL1 plus certaines extensions), s'exécutant sous le système VM/370/CMS. Le programme et les données sont vus comme des suites de structures de phrases et d'unités lexicales. L'affichage des programmes est guidé par la grammaire du langage et l'affichage des données est dépendant de leurs déclarations.

Les outils constituant l'environnement de programmation sont :

- un décompilateur,
- un éditeur,
- un interpréteur,
- un compilateur.

Ces différents outils dialoguent avec l'utilisateur au travers d'une interface unique. Nous allons tout d'abord décrire

l'interface d'affichage, puis nous détaillons les trois outils énumérés.

4.1.1. Interface utilisateur

Chaque session communique avec l'utilisateur au travers d'un terminal virtuel qui est projeté sur un terminal réel ou sur une partie de celui-ci. Chaque terminal virtuel est constitué de deux zones :

- une zone formattée qui contient les programmes,
- une zone de messages, dans laquelle s'effectue le dialogue avec l'utilisateur.

L'affichage du programme est réalisé par un paragrapheur ou décompilateur ; il transforme la représentation intermédiaire en texte source. Chaque phrase peut être découpée en sous-phrases ; l'indentation se fait en fonction du niveau d'imbrication des phrases ; si le niveau est trop élevé, la chaine '....' se substitue au texte réel. Le pointeur d'exécution est matérialisé par une mise en surbrillance de l'instruction correspondante. L'exemple qui suit illustre le découpage en sous-phrases :

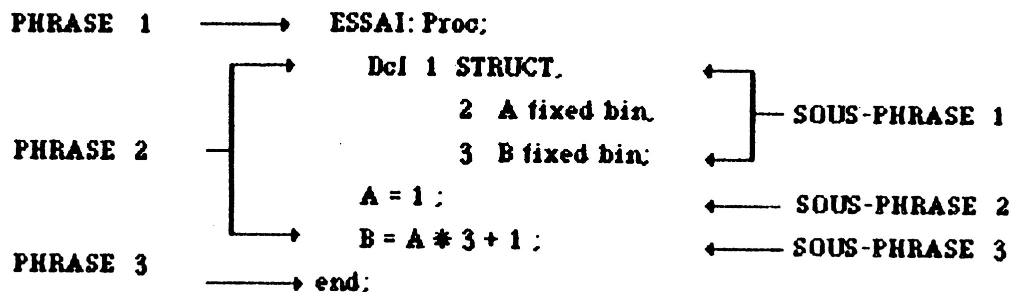


FIGURE 4

Plusieurs commandes manipulent le point d'affichage courant. La commande DELETE détruit la phrase correspondante et déplace le point courant, appelé "focus" sur la phrase suivante. Un focus est une phrase mise en valeur. Toutes les manipulations se font par rapport au focus ou par rapport à un nom associé à un ensemble de focus. L'utilisateur peut donner un nom à un ensemble de phrases ;

la mémorisation de ce nom est alors faite dans un objet appelé TAG LIST. L'utilisateur peut demander au système de mettre en évidence toutes les phrases marquées.

Plusieurs sessions peuvent coexister en parallèle ; chaque session utilise une partie de l'écran pour montrer son état courant ; elles communiquent avec l'utilisateur par l'intermédiaire du terminal virtuel correspondant. Plusieurs fenêtres sont actives à un instant donné mais une seule est courante, c'est à dire que les commandes s'adressent par défaut à cette fenêtre.

Les différentes sessions actives constituent un espace de travail (WS) qui peut être sauvegardé sous un nom donné et rappelé par la suite.

Le WS est en effet un module exécutable (pas dans son format de stockage) du système d'exploitation de la machine hôte. Ceci implique que l'utilisateur n'a pas à connaître le système hôte, mis à part pour lancer PDE.

4.1.2. Editeur

L'éditeur permet de manipuler des objets structurés de différents types.

Tous les objets de PDE sont vus comme des structures arborescentes de phrases et d'unités lexicales. Les unités lexicales sont atomiques et sont donc des composants indivisibles d'un objet.

Si une procédure P2 est interne à P1, PDE traite le texte de P2 comme un objet séparé ; P2 peut être éditée séparément mais reste toujours liée à P1 (par les variables globales) ; ceci implique que les variables déclarées dans P1 soient connues par P2. Une procédure peut également être complètement détachée de la procédure qui la contient ; dans ce cas, elle devient un fragment de procédure qui peut alors contenir des variables non déclarées et des attributs non résolus.

4.1.3. Commandes de mise au point

Les données mémorisées dans des variables sont accessibles seulement en PL1L, c'est à dire par des instructions manipulant des variables. Comme résultat, les objets "données" héritent de la structure de la déclaration associée à la variable. Les phrases de données sont des tableaux de sous-structures. Les fichiers de données sont structurés en vecteurs d'enregistrements. Les fichiers internes sont des objets générés par PDE et se comportent comme des fichiers normaux tels que le programme concerné.

On peut exécuter une instruction ou une expression à n'importe quel point de la session ; l'exécution et le résultat de l'exécution dépendent du contexte courant. L'environnement de données est marqué à "non initialisé" en début de session pour les variables accessibles. Si une variable non initialisée est utilisée dans une expression, l'utilisateur doit donner une valeur à cette variable ou arrêter l'exécution. Toute affectation persiste tout au long de la session.

L'exécution peut être continue ou interactive ; dans le premier cas, le programme s'exécutera jusqu'à ce qu'une condition d'arrêt soit remplie (Point d'arrêt, Fin du programme, erreur, etc).
Un point d'arrêt peut être défini soit comme une "place" (une instruction vide) dans une procédure soit comme un prédicat ; la reprise de l'exécution s'effectue dans les deux cas sur l'instruction placée immédiatement après. L'exécution peut par ailleurs se faire en pas à pas ou par groupe d'instructions.

La modification d'une procédure active provoque l'annulation de la chaîne d'appels créée à partir de cette procédure ; l'exécution ne pourra alors reprendre qu'au point d'entrée de cette procédure. Il faut indiquer que l'environ statique d'une procédure est sauvegardé systématiquement à chaque appel de celle-ci.

Le nombre de commandes de mise au point est faible, mais certaines commandes permettent de réaliser des fonctions

complexes. L'exécution d'un programme ou d'une procédure peut être lancée par l'une des commandes suivantes.

a) START <nom_procédure> <arguments>

La procédure (ou le programme) donnée en paramètre est alors activée à son point d'entrée avec les paramètres définis par le deuxième paramètre.

b) RUN <paramètres>

<paramètres> désigne dans ce cas un ensemble d'options pour cette commande. Suivant leur valeur, l'exécution peut être relancée plusieurs fois, mais les points d'arrêt sont ignorés.

c) CONTINUE <paramètre>

L'exécution est relancée au point courant jusqu'à ce que la condition donnée en paramètre soit remplie. Il est possible de relancer le programme en ignorant les arrêts conditionnels ou les traces.

d) STEP N

Cette commande permet de lancer l'exécution du programme en mode pas à pas. Le pas est donné par le paramètre qui suit la commande, lequel indique le nombre d'instructions simples à exécuter.

La commande suivante permet d'évaluer une expression ou une instruction et donc de visualiser une variable ou d'affecter une valeur à une variable donnée.

VALUE <instruction> ou VALUE <expression>

D'autres commandes permettent de manipuler des points d'arrêt et d'activer des traces de variables :

a) BREAK <paramètres>

On peut, par cette commande positionner des points d'arrêt qui peuvent être conditionnels ou non. Certains paramètres permettent également de lancer l'exécution en mode interactif.

b) BREAKS <paramètre>

Cette commande permet de lister les points d'arrêt d'une procédure, dont le nom est donné en paramètre, ou de lister toutes les procédures contenant des points d'arrêt.

c) CHECK <paramètres>

Cette commande est utilisé pour tracer ou arrêter la trace d'une variable donnée ; cette trace peut être conditionnelle. Elle peut également servir pour positionner un point d'arrêt.

Conclusion

Le système PDE1L est le système qui nous a le plus séduit. Nous ne lui ferons que quelques reproches : aucune réelle visualisation du programme n'est faite en mode dit interactif au cours de l'exécution, et les différentes possibilités offertes par une même commande ne sont assimilables que par beaucoup de pratique.

4.2. Le systéme INTERLISP

Voulant décrire un système LISP, notre choix s'est tourné vers ce système, car il offre de nombreuses fonctions intéressantes et en particulier des outils de mise au point.

INTERLISP est un outil destiné à faciliter la production de programmes LISP. Le dialecte INTERLISP offre les fonctions de base LISP classiques, plus un ensemble de fonctions permettant d'écrire des instructions du type IF-THEN-ELSE et FOR-WHILE-DO. Il permet

également l'utilisation d'opérateurs infixés tels que : "+", "-", "*", "/", "=", etc.

Le système comprend un éditeur syntaxique permettant de créer et de modifier des fonctions LISP, et un interpréteur de ce langage ; un ensemble de fonctions de mise au point est fourni avec le système. Nous ne détaillons ci-après qu'une partie des fonctions offertes, car celles-ci sont trop nombreuses et souvent propres à LISP.

4.2.1. Fonction de trace

Cette fonction permet de tracer l'appel d'une fonction. A chaque appel de celle-ci, la trace imprime le nom de la fonction, le nom et la valeur de chacun de ses paramètres.

Pour les fonctions appelées récursivement, chaque appel est paragraphé. Soit la fonction factorielle définie de la façon suivante :

```
Factorielle (N) :
   Si N = 0 ALORS Factorielle = 1
            SINON Factorielle = N * Factorielle (N - 1)
Fin Factorielle
```

La trace de l'exécution de l'appel Factorielle (4) serait la suivante :

```
   Factorielle:
    N = 4
      Factorielle:
       N = 3
         Factorielle:
          N = 2
            Factorielle:
             N = 1
               Factorielle:
                N = 0
               Factorielle = 1
            Factorielle = 1
         Factorielle = 2
       Factorielle = 6
   Factorielle = 24
24
```

4.2.2. Arrêts conditionnels

Un point d'arrêt conditionnel peut être inséré au niveau d'une étiquette placée devant une instruction. L'exécution s'arrête sur cette étiquette si la condition donnée est vérifiée ; il est alors possible d'exécuter n'importe quelle fonction du système.
On peut ensuite relancer l'exécution de différentes façons. Le premier moyen est de relancer le programme en supprimant le point d'arrêt et en réaffichant le résultat de l'expression associée au point d'arrêt. Une des autres possibilités consiste à relancer l'exécution en laissant le point d'arrêt.

4.2.3. Retour arrière

Lorsqu'une erreur d'exécution se produit, on peut réexécuter la fonction qui a produit l'erreur en réinitialisant l'environnement à l'état de l'appel. Tout effet de bord susceptible de s'être produit lors de cette dernière exécution est ainsi neutralisé. On peut alors positionner des points d'arrêts

conditionnels dans cette fonction pour vérifier la valeur des différentes variables et détecter ainsi l'erreur.

De plus, il est possible de visualiser les variables locales de toute fonction apparaissant dans la pile d'exécution. Il suffit de déplacer le pointeur de pile au niveau désiré et de donner le nom des variables que l'on veut consulter.

On peut également imprimer le contenu de la pile. Dans ce cas, on obtient le nom des fonctions actives ainsi que leur niveau d'exécution et la valeur de leurs paramètres au moment de l'appel.

Conclusion

Ce système nous a paru très intéressant sur trois points précis. Nous avons apprécié tout particulièrement les possibilités offertes par les traces ainsi que la possibilité de réexécuter après une erreur d'exécution la dernière fonction appelée. Une autre fonction intéressante est la visualisation de la pile avec la valeur des paramètres des fonctions actives. Enfin, le fait de pouvoir accéder aux différentes variables locales d'une fonction est très utile pour pouvoir bien examiner le déroulement du programme. Notons également qu'il est possible de modifier une fonction sous l'éditeur et de continuer ensuite l'exécution.

Nous regrettons toutefois qu'il n'existe pas d'exécution pas à pas ni les mécanismes nécessaires à sa mise en oeuvre.

4.3. DICE

DICE (Distributed Incremental Compiling Environment) (<Fritzson 82-83>) est un environnement de programmation PASCAL développé par une équipe universitaire suédoise. Cet environnement est centré sur une représentation arborescente des programmes sources et est composé des outils suivants :

- un compilateur incrémental,
- un éditeur structuré et un éditeur de textes couplé à l'analyseur syntaxique du compilateur incrémental,
- un éditeur de liens dynamique,

- une base de données,
- un metteur au point.

L'originalité de cet environnement est l'utilisation d'un compilateur incrémental et d'une base de données dont l'objectif principal est l'aide à la mise au point ; ces outils produisent et conservent toutes les informations nécessaires au metteur au point. DICE s'exécute sur une machine hôte (DEC 20) et génère du code pour une autre machine (PDP 11) connectée au calculateur hôte par un réseau local.

La compilation se fait à partir d'une représentation arborescente des programmes. Le rôle du metteur au point est de transformer les instructions de mise au point rajoutées par l'utilisateur en un ensemble d'appels de procédure. Ces appels sont insérés dans le programme source par l'utilisateur en suivant la syntaxe du langage de programmation et sont traduits par l'éditeur ; l'éditeur est ensuite activé pour traiter ces instructions. Toutes les instructions PASCAL peuvent être considérées comme des commandes par le metteur au point. Nous allons voir dans le schéma suivant les relations entre la représentation interne et le code généré.

Soit l'instruction PASCAL suivante :

```
WHILE i < 15 DO i := i + 2 ;
```

Cette instruction est compilée de la façon suivante :

```
Instr_While :
     Comparer i à 15
     Si i >= 15 alors aller à Fin_While
     ajouter 2 à i
     aller a Instr_While
Fin_While :
```

Deux attributs sont attachés à certains noeuds de l'arbre. Le premier indique la taille de l'instruction et le second, manipulé

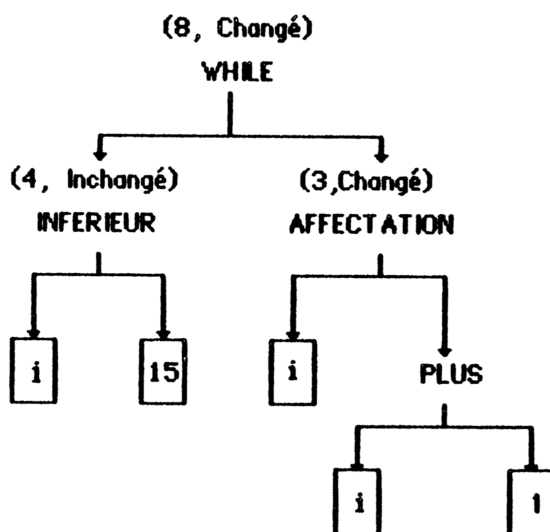
par l'éditeur, indique au compilateur s'il y a eu modification dans ce sous-arbre.

Supposons que l'utilisateur modifie l'instruction d'affectation de l'exemple précédent. La nouvelle instruction WHILE devient :

```
WHILE i < 15 DO i := i + 1 ;
```

Le schéma suivant montre les modifications apportées à l'arbre et celles du code généré qui en découle.

Arbre après modification par l'éditeur



Code PDP11 pour l'affectation :
i := i + 2

4	1: 2: CMP -2(R5), 15 3: 4: BGT +5
3	5: 6: ADD 2, -2(R5) 7:
1	8: BR -7

Arbre après recompilation incrémentale

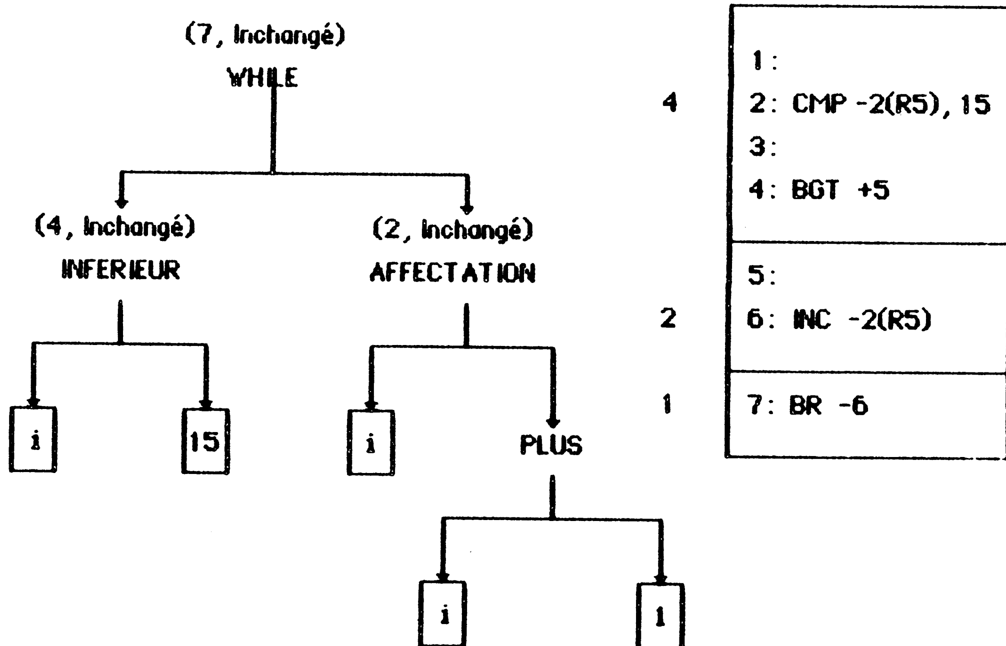


FIGURE 5

Comme les instructions de mise au point sont des instructions PASCAL, seules trois commandes supplémentaires ont été rajoutées pour la mise au point. Elles permettent de :

- Tracer les variables.
- Exécuter le programme en pas à pas.
- Placer des points d'arrêt conditionnels.

4.3.1. Trace de variables

Lorsqu'une commande de trace est soumise par l'utilisateur, le système va chercher dans la base de données toutes les occurrences de la variable désignée. Si la position de la variable est telle qu'elle peut être modifiée (affectation, instruction FOR, paramètre par référence), une instruction WRITE (Variable) est insérée après l'instruction. Une fois le programme modifié, le metteur au point relance le compilateur. Si le programme contient des variables cachées, le système demande à l'utilisateur s'il

veut tracer les différentes variables de même nom.

4.3.2. Pas à pas

Pour réaliser le pas à pas, le metteur au point insère un point d'arrêt avant la prochaine instruction à exécuter et supprime le point d'arrêt courant.

4.3.3. Points d'arrêt conditionnels

Un point d'arrêt conditionnel est transformé en une instruction IF PASCAL ; cette instruction est insérée à l'endroit désigné.

Conclusion

Les avantages d'un tel système sont nombreux. En effet, la réalisation du metteur au point est simplifiée. L'adjonction de commandes est relativement simple car il suffit de transformer la nouvelle commande en instruction PASCAL. L'exécution du programme en mode mise au point est rapide car l'interprétation du code correspond à l'exécution du code compilé. De plus, il est possible de relancer le programme à l'endroit courant après correction de l'erreur.

Notons, toutefois, qu'il est difficile de réaliser une visualisation de l'exécution sur l'écran. D'autre part, le fait d'exécuter des instructions PASCAL ne permet pas d'effectuer tous les tests de mise au point (variable affectée ou non, ...). De plus, le fait d'accéder aux variables par des instructions PASCAL ne permet pas de manipuler toutes les variables actives du programme à un instant donné : variables globales masquées par des variables locales ou variables locales de procédures appelées récursivement.

4.4. CPDS

CPDS (Cornell Program Development System : <Teitelbaum81>) est un environnement de programmation basé sur un éditeur syntaxique et permettant de créer, d'éditer, d'exécuter et de mettre au point des programmes. Il a été développé à l'Université de Cornell (New-York), et est destiné à l'apprentissage du langage PL/CS (sous-ensemble du langage PL/1) par les étudiants de cette université. Il a été implémenté sur des LSI-11 fonctionnant sous le système UNIX.

L'ensemble des outils de l'atelier manipulent une structure arborescente où chaque noeud de l'arbre à la structure suivante :

TYPE INSTRUCTION	INDICATEURS	INSTRUCTION PRECEDENTE	EXECUTION SUIVANTE	NOMBRE D' INSTRUCTIONS	ORDRES
		INSTRUCTION SUIVANTE			

FIGURE 6

Le champ "Type Instruction" identifie l'instruction à décoder. Il comprend en outre des renseignements supplémentaires permettant la décompilation du programme. Certains noeuds de l'arbre ne sont pas décompilables mais sont rajoutés pour marquer la fin des blocs d'instructions et d'autres fins d'instructions.

Le champ "Indicateurs" constitué d'un ensemble de booléens, permet de préciser la façon dont le programme sera affiché, les erreurs sémantiques de l'instruction et les différentes indications pour l'exécution du programme.

Les pointeurs "Instruction précédente" et "Instruction suivante" servent à localiser l'instruction dans le texte source. Le deuxième chaînage permet également de suivre le flot d'exécution du programme.

Le pointeur "Exécution suivante" n'existe que si cette

instruction est conditionnelle (FOR, IF, ...) et précise l'instruction à exécuter si l'évaluation de la condition de l'instruction est fausse. Si l'expression est vraie, l'exécution se poursuivra à l'instruction correspondant au pointeur "instruction suivante".

Le champ "Nombre d'instructions" indique le nombre d'instructions élémentaires ("action routines") composant l'instruction. Par exemple, une instruction "FOR" PASCAL est composée de trois instructions élémentaires :

- Initialisation de la variable de boucle.
- Incrémentation de cette variable.
- Test de la valeur de cette variable.

Le schéma suivant montre un programme PL/1 et l'arbre associé.

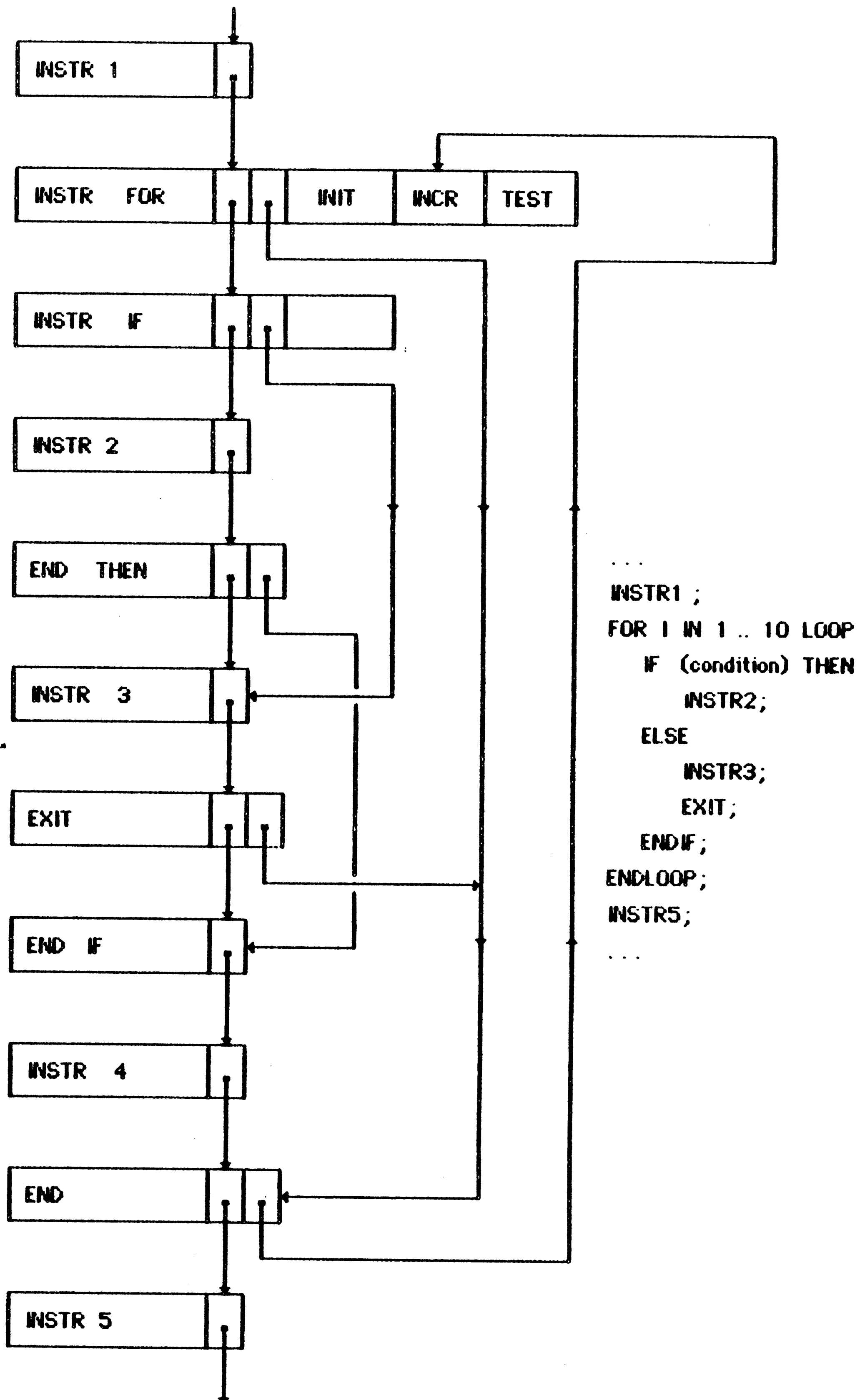


FIGURE 7

4.4.1. Commandes de mise au point

Les commandes de mise au point sont regroupés en trois classes :

a) Commandes d'exécution.

L'exécution d'un programme peut se faire soit en mode normal, soit en mode mise au point. Le mode normal permet de ne pas prendre en compte toutes les fonctions de trace, de pas à pas et les points d'arrêt. L'exécution du programme peut se faire en mode visuel ou en mode normal. Dans le premier cas, la vitesse d'exécution est un paramètre de la fonction, le curseur est placé sur l'instruction à exécuter et une temporisation est faite entre chaque instruction.
Il est également possible d'exécuter le programme en mode pas à pas. Le pas de programme est fixe et correspond à une instruction simple du langage.

b) Visualisation des variables

Deux fonctions permettent d'accéder à la valeur des variables. La première permet un accès statique et ne peut être activée que si le programme est arrêté ; la seconde est une visualisation dynamique et correspond à la fonction de trace qui existe sur un grand nombre de metteurs au point. Dans ce dernier cas, une fenêtre sur l'écran est allouée à cette fonction ; les noms des variables scalaires apparaissent dans cette fenêtre ainsi que la valeur qui leur est associée. On voit ainsi évoluer le programme (si celui-ci a été lancé en mode visuel) en même temps que les variables.

c) Formattage de l'écran

Cette fonction permet de modifier la taille des différentes fenêtres de l'écran. Lors d'une session de mise au point, l'écran est divisé en trois fenêtres.

La première est destinée à contenir la partie du programme active. La seconde recevra les sorties effectuées par le programme en cours d'exécution et la troisième est destinée au traces des variables.
A ces trois fenêtres s'ajoute une fenêtre fixe permettant d'entrer les commandes.

Conclusion

Un tel système est intéressant sur plusieurs point. La structure de la représentation interne permet de stocker les programmes écrits en différents langages si ceux-ci sont conformes à une structure de blocs. L'interpréteur est commun à tous ces langages, et il n'est donc pas nécessaire d'en écrire un pour les différents formalismes.
La visualisation du programme à vitesse variable est aussi une fonctionnalité très intéressante, car elle permet de bien suivre l'exécution de son programme, et évite par là même d'exécuter certaines parties en mode pas à pas.
La façon dont est implémentée la trace des variables limite le bruit visuel qui aurait été occasionné par le défilement de valeurs associée à une variable, et permet de se concentrer sur l'évolution de son programme.

Il faut cependant regretter que seules les variables scalaires soient visualisables. De plus il est impossible de modifier la valeur d'une variable, ce qui peut être très utile pour continuer un test lorsqu'on s'aperçoit que la valeur d'une variable est erronée ou non initialisée.

5. CONCLUSION

Pour résumer cette étude bibliographique nous regroupons dans le tableau qui suit, les différents systèmes étudiés. Ce tableau comprend deux parties :

- les quatre premières lignes résument les caractéristiques principales de ces metteurs au point,
- les quatorze suivantes contiennent les fonctions de mise au point représentatives rencontrées dans les différents systèmes.

La dernière colonne du tableau montre les fonctions retenues pour le metteur au point réalisé dans le cadre du projet ADELE.

Pour des raisons de présentation, nous condensons les libellés des fonctions et nous plaçons une étoile lorsque le système offre cette possibilité.

Nous explicitons ici les libellés correspondants aux numéros suivants :

3- Caractéristique du système précisant que celui-ci est indépendant du langage de programmation.

4- Les instructions de mise au point sont des instructions du langage de programmation.

5- Il y a possibilité de suivre sur l'écran le pointeur d'exécution lorsque le programme s'exécute.

6- L'exécution peut continuer après une modification du code source.

7- L'exécution peut se poursuivre à un endroit différent du point courant.

8- Il est possible de réexécuter un bloc en réinitialisant les différents environnements englobants aux valeurs de l'appel précédent.

	SYSTEME / FONCTIONS	PERQ	ADB	PROBE	PDE 1L	INTER-LISP	DICE	CPDS	ADELE
1	Mise au point avec interpréteur				*	*		*	*
2	Mise au point sur code compilé	*	*	*	*		*		
3	Multiformalisme	*	*	*				*	
4	Instr. MisePoint en Lang. de Prog.					*	*		
5	Exéc. Visuelle							*	*
6	Cont. Exéc. après Modif. du code		*		*	*	*	*	*
7	Suite Exéc. autre Instruction		*	*			*	*	*
8	Retour Arrière avec Contexte Init.				*	*			
9	Pas à Pas		*	*	*		*	*	*
10	Paramétrisation Valeur Pas		*	*	*				*
11	Points d'arrêt		*	*	*	*	*	*	*
12	Test Variable initialisée				*	*		*	*
13	Accès Variables par nom			*	*	*	*	*	*
14	Accès à toutes les variables	*	*			*			*
15	Modif. contenu variables		*	*	*	*	*		*
16	Trace Variables		*	*	*	*	*	*	*
17	Trace Fonctions			*	*	*	*	*	*
18	Visu. Pile des appels Procédures	*	*	*		*		*	*

CHAPITRE 3

UNE SOLUTION A LA MISE AU POINT

1. CADRE DE TRAVAIL

ADELE (Atelier de DEveloppement de LogiciEL) (<Briat 81>,<Estublier 83>) est un environnement de programmation pour PASCAL, composé de différents outils qui travaillent sur une représentation intermédiaire unique (<Mossière 82>). ADELE est destiné à de petites équipes de recherche développant des applications généralement complexes, de taille moyenne (quelques dizaines de milliers de lignes) et ne nécessitant que peu de maintenance. Cet environnement s'exécute sur des postes individuels, connectés à un ordinateur central (CII-HB 68 DPS 8 sous le système MULTICS). Dans la première version du système, les postes de travail sont des terminaux du type VT100. Toutefois, il est prévu dans les objectifs du projet de les remplacer par des micro-ordinateurs 16 bits offrant des systèmes de visualisation graphique du type "Bit-Map" et des dispositifs de désignation évolués (souris, "Joysticks", etc).

Les machines visées actuellement sont le SM90 (<Finger 82>) ou le Micromèga (<Thomson 82>).

La représentation interne d'ADELE est calquée sur l'arbre abstrait du programme. Cet arbre est également décoré par des attributs sémantiques. Les différents outils manipulant cette représentation interne sont :

- un éditeur syntaxique,
- un décompilateur,
- un interpréteur/metteur au point,
- un générateur de code,
- un introducteur.

Tous ces outils dialoguent avec l'utilisateur au travers d'une

interface usager, permettant d'unifier le dialogue utilisateur. Les programmes sont archivés par un système de gestion de base de programmes. Cet atelier peut être schématisé par la figure 8 :

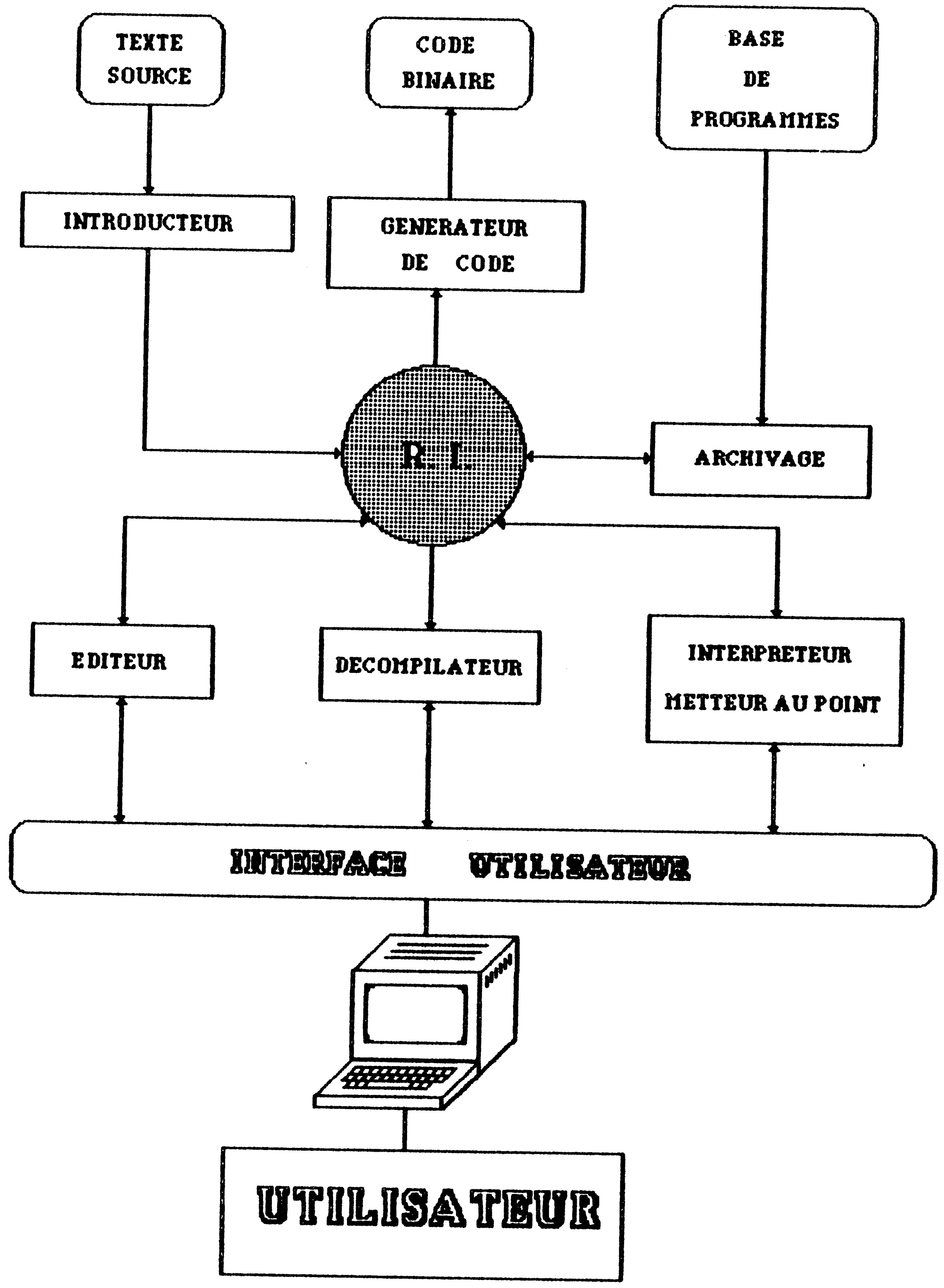


FIGURE 8

Nous allons maintenant détailler les différents outils de l'atelier.

1.1. Le système d'archivage

Ce système gère une base de programmes. Il permet d'extraire ou d'archiver les différents composants d'un programme appelés modules. Un module est l'association de deux composantes (<Cheval 82>) : l'interface et le corps.
L'interface, décrit les ressources fournies par le module ; le corps réalise effectivement ces ressources. Concrètement les ressources sont des constantes, des types, des variables ou des procédures. Pour celà, un certain nombre d'extensions au PASCAL langage standard ont été réalisées.
Ce système permet également de vérifier la cohérence entre les différents modules (<Ghoul 83>) et de gèrer les différentes versions d'un module (<Estublier 83b>). Ce gestionnaire peut ainsi fournir différents programmes dont la fonctionnalité est identique, mais la réalisation différente.

1.2. L'introducteur

L'objectif de cet outil est de traduire des programmes PASCAL source dans la représentation interne d'ADELE. Il permet par conséquent d'introduire des programmes PASCAL existants dans l'atelier.

1.3. Le générateur de code

Il s'agit en réalité d'un système de production automatique de générateurs de code (<Cassagne 82>,<Santana 83>) ; la construction d'un générateur se fait à partir d'une description de la machine cible, décrite dans un langage fourni par le système. La génération de code s'effectue à partir de la RI et produit en sortie du code objet pouvant être traité par l'éditeur de liens de la machine cible.

1.4. L'éditeur syntaxique

Cet éditeur syntaxique (<Rouzaud 84>) permet de construire et de modifier des programmes. La construction de programme est guidée par la grammaire du langage et s'effectue par remplissage de "boites" générées automatiquement et appelées métavariables (instructions, identificateur, expression, etc.). Toutes les données entrées sont vérifiées immédiatement, tant sur le plan syntaxique que sémantique.

1.5. L'interface utilisateur

Elle est chargée, d'une part de gérer le dialogue entre l'utilisateur et l'application, et d'autre part, d'afficher les objets manipulés par l'utilisateur. Cette interface est réalisée par deux outils : le médiateur et le compositeur.
Le médiateur (<Coutaz 83>) gère le dialogue avec l'utilisateur. Les commandes entrées sont analysées puis ventilées vers l'outil correspondant dans la syntaxe propre à cet outil.
Le compositeur (<Herrmann 82>) travaille sur une arborescence de boites créée par le décompilateur. Cet arbre contient des liens vers la représentation interne. Ces liens permettent d'associer une instruction RI à sa forme décompilée. Ce compositeur met en forme sur l'écran les différentes informations de cet arbre, et permet la navigation dans l'arbre, et par là même, dans la représentation interne.

La représentation interne des programmes et l'interpréteur-metteur au point font l'objet des chapitres suivants.

1.6. Présentation des fonctionnalités du metteur au point

Nous avons décrit sommairement les différentes méthodes de mise au point utilisées à l'heure actuelle. Ci-après, nous présentons brièvement les possibilités offertes par notre système de mise au point.

L'architecture de l'atelier nous imposait une représentation interne des programmes et nous a amenés à réaliser un interpréteur. Le metteur au point exploite au maximum les possibilités de l'interprétation et doit être par conséquent, tout au moins pour ses fonctionnalités externes, considéré comme un unique outil pour l'utilisateur. Les différentes fonctions que nous avons retenues pour notre metteur au point sont les suivantes :

a) Exécution en pas à pas

L'utilisateur peut exécuter ses programmes de deux façons différentes. Tout d'abord l'exécution en mode normal qui correspond à une exécution séquentielle jusqu'à la rencontre d'une condition d'arrêt (fin du programme, point d'arrêt, erreur d'exécution, etc). La seconde est une exécution pas à pas. Dans ce mode, deux possibilités sont offertes ; la première est le pas à pas simple avec paramétrisation de la valeur du pas, la seconde est le pas à pas "automatique" avec réglage de la temporisation aprés affichage de l'instruction venant d'être exécutée.

b) Insertion de points d'arrêt.

L'utilisateur peut positionner des points d'arrêt, avant ou après n'importe quelle instruction de son programme. Un point d'arrêt est inconditionnel mais pondéré par une valeur entière qui spécifie le nombre de passages sur l'instruction à partir duquel doit être réalisé l'arrêt.

c) Manipulation des variables

Toutes les variables actives du programme peuvent être consultées ou modifiées. L'accès à ces variables ne se fait pas par des instructions PASCAL, mais par des commandes spécifiques permettant d'augmenter la puissance d'expression de cette fonction ; une variable est désignée par son nom

PASCAL. Nous permettons à l'utilisateur de visualiser le contenu de la variable désignée, ainsi que de modifier la valeur qui lui est associée ; le résultat est affiché conformément au type de déclaration de la variable. Les variables peuvent également être tracées durant l'exécution du programme. Une autre fonction offerte est la protection de variables en écriture ; dans ce cas, lorsqu'une instruction du programme tente de modifier la variable protégée, l'exécution est arrêtée.

d) Visualisation de la pile d'appels

Les procédures ou fonctions du programme actives à un instant donné peuvent être affichées suivant l'ordre d'appels.

e) Exécution d'un fichier de commandes

On peut demander à l'interpréteur d'exécuter un ensemble de commandes de mise au point quand il rencontre une condition d'arrêt. Ces commandes sont rassemblées dans un fichier spécialisé (contexte), créé par l'utilisateur.

La suite de cet exposé décrit l'interpréteur du système ADELE et détaille l'ensemble des commandes offertes.

2. REPRESENTATION INTERNE

La représentation interne (R.I) d'un programme PASCAL est la forme unique de manipulation d'un programme par les outils de l'atelier. Cette représentation est une représentation chaînée que l'on peut parcourir de façon séquentielle ou quelconque. Elle contient les informations du programme, de type syntaxique (pour l'édition), et également de type contextuel (analyse des identificateurs pour l'interprétation et la génération de code). Ceci implique que le squelette de la R.I est un arbre calqué sur l'arbre abstrait du langage. Chaque sommet de l'arbre correspond à une unité syntaxique et comporte des attributs de natures différentes :

- Les attributs structurels dénotent les fils syntaxiques du sommet considéré (ex : les fils condition, partie-then, partie-else d'un sommet instr-if) ; ce sont des arcs vers les sommets racines des sous-arbres fils correspondants ;
- Les attributs lexicaux mémorisent les chaînes de caractères associées à un identificateur, une constante, etc.
- Les attributs contextuels traduisent l'insertion d'un sommet dans son contexte : lien entre occurrence d'identificateur et sa définition, type d'une expression, erreurs rencontrées, etc.

Ces attributs sont le plus souvent des arcs vers d'autres sommets.

La R.I est donc un graphe, mais la notion de squelette arborescent (donné par les attributs structurels) offre une meilleure compréhension de la structure de données.

Prenons le programme suivant pour illustrer toutes ces définitions :

```
PROGRAM p (input, output) ;
   VAR i, j : INTEGER ;
   BEGIN
     READ (i, j) ;
     IF i > j
        THEN i := i - j
        ELSE writeln (j)
   END.
```

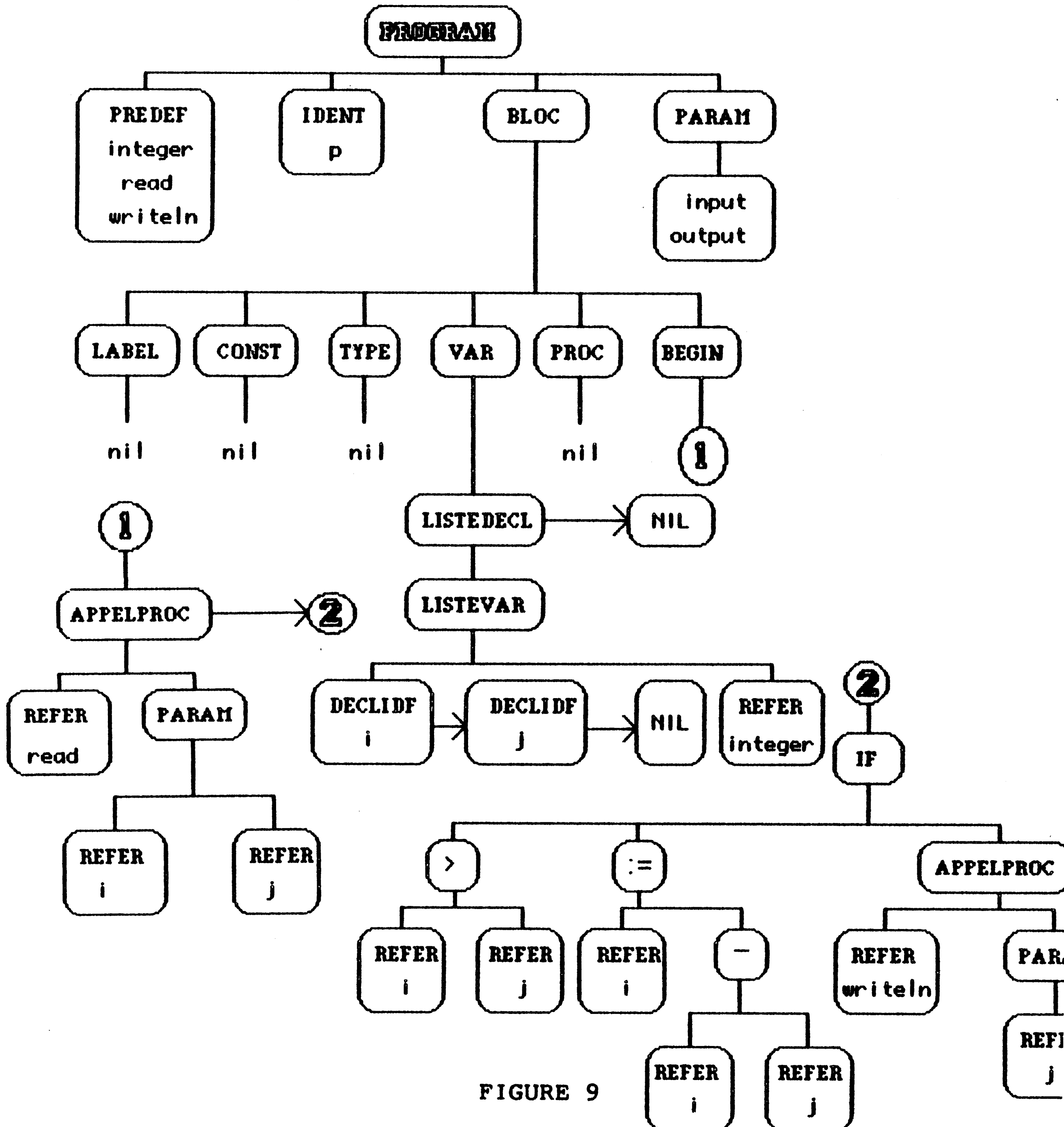


FIGURE 9

La figure 9 montre la structure de la représentation interne associée au programme donné en exemple. Elle met en évidence les différents liens existants dans cette R.I.

Un ensemble de procédures de manipulation de la R.I. sont fournies avec la définition de celle-ci ; elles permettent de masquer les détails d'implémentation et de se placer à un niveau supérieur. Ces procédures peuvent être classées en trois groupes. Le premier correspond aux fonctions de déplacement dans l'arbre : père, fils, frère, etc. Le deuxième groupe est constitué des fonctions de manipulation des attributs : typesommet, chaine, max, min, declar, etc. ; enfin, les fonctions du dernier groupe permettent de manipuler des sous-arbres : détruire, lier, insérer, etc. Ces différentes fonctions de manipulation sont données en annexe avec la description complète de la R.I.

Dans ce qui suit, nous décrivons brièvement certains attributs utilisés par l'interpréteur. Nous nous limitons aux attributs communs à différents noeuds, les attributs spécifiques étant détaillés lors de leur utilisation.

A chaque noeud de l'arbre est associé un attribut erreur, précisant la présence éventuelle d'une erreur dans le sous-arbre. Le type d'erreur peut être de différentes natures (arbre incomplet, erreur sémantique).

Sur les noeuds référençant un identificateur, un attribut lie cet identificateur avec la déclaration correspondante, compte tenu des règles de portée et de visibilité.

Sur une affectation, un attribut pointe éventuellement sur un sous-arbre exprimant une contrainte associée à cette affectation : réduction d'un intervalle d'entiers, restriction d'ensemble, etc. Cette contrainte peut être évaluée en consultant les attributs min et max qui lui sont affectés. Par ailleurs, étant donné que PASCAL permet d'écrire des constructions du style suivant :

```
type
   T1 : integer;
   t2 : T1;
   T3 : T2;
var
   I : T3 ;
       .......
```

il est nécessaire, pour accélérer la recherche du type de base d'un identificateur, d'avoir un chaînage direct vers celui-ci. Pour celà, un attribut permet de court-circuiter les différentes indirections en désignant directement le type de base (attribut base).

3. L'INTERPRETEUR

Nous présentons dans ce paragraphe les différents composants de l'interpréteur et le traitement des différents objets manipulés par celui-ci.

3.1. Structure des programmes PASCAL

Le langage PASCAL offre une structure de blocs. Un bloc est ouvert par la déclaration d'une procédure. Une telle structure permet de définir des procédures imbriquées, utilisant des objets propres (types, variables, etc). Il est aussi possible de redéfinir des objets définis dans des procédures englobantes. La durée de vie de ces objets, ainsi que leur portée sont définies par Aho et Ullman (<Aho 79>) :

- Un objet déclaré dans un bloc n'est utilisable qu'à l'intérieur de ce bloc.
- Si un bloc B est imbriqué dans un bloc A, tout objet utilisable dans A est aussi utilisable dans B, à moins que l'identificateur de cet objet n'ait été redéclaré dans B.

Les règles de visibilité définies engendrent une structure

arborescente de l'imbrication des procédures. Dans l'arbre ainsi constitué, un noeud représente une procédure et un lien une relation d'imbrication. Tout ceci constitue la structure statique d'un programme. Le niveau du noeud de l'arbre représentant une procédure est appelé le niveau statique de cette procédure.

La figure suivante met en évidence la structure de bloc et fait apparaitre les différents niveaux statiques d'un programme PASCAL.

```
PROGRAM niveau ;                     Niveau 0     Programme Niveau
                                                         !
   VAR i, j, k : INTEGER ;                       ________!________
                                                 !               !
   PROCEDURE a (s : INTEGER);        Niveau 1   Procédure A   Procédure C
                                                 !
      VAR i : INTEGER ;                          !
                                                 !
      PROCEDURE b ;                  Niveau 2   Procédure B
         VAR j : INTEGER ;
         BEGIN
          (* Corps Proc b *)
         END ; (* Fin Proc b *)
      BEGIN
       (* Corps Proc a *)
      END ; (* Fin Proc a *)
   PROCEDURE c ;
      VAR k : INTEGER ;
      BEGIN
       (* Corps Proc c *)
      END ; (*Fin Proc c *)
  BEGIN
   (* Corps Prog Principal *)
  END. (* Fin prog Principal *)
```

3.2. Allocation de variables

L'allocation de variable est la fonction qui permet d'attribuer à chaque variable du programme une place en mémoire. Cette place est mémorisée par une adresse constituée de deux parties : l'adresse de début de l'environ local et le déplacement par rapport à cet environ. L'allocation de variable est répartie dans le temps. Une première étape d'allocation est réalisée au lancement de l'interpréteur en vue de réserver la place nécessaire aux variables globales (niveau zéro). Puis, une nouvelle allocation est faite à chaque activation d'une procédure ou fonction ; la place correspondant à cette allocation est récupérée en fin d'exécution.

L'interpréteur met en oeuvre une mémoire qui lui est propre. Son organisation est similaire à la mémoire d'une machine, c'est à dire qu'elle peut être considérée comme un ensemble de cellules de base, chaque cellule étant l'unité d'allocation minimale.

Notre technique d'allocation n'est pas optimale en place, mais l'accès aux données est optimal en temps car il ne nécessite pas de compactage ni de décompactage de l'information. A chaque élément de base constituant une variable sont associés trois indicateurs permettant d'effectuer certains contrôles à l'exécution. Ces trois indicateurs sont initialisés à faux lors de l'initialisation de l'interpréteur.

Le premier permet de savoir si la variable a été initialisée par une instruction (affectation, lecture d'un fichier, for, new, etc).

Le deuxième indicateur est utilisé en mise au point. En cours d'exécution, il est possible de visualiser la valeur d'une variable (trace) chaque fois que celle-ci est modifiée. C'est donc sur une instruction de mise au point que cet indicateur est modifié.

Enfin, le dernier indicateur est commun à l'interpréteur et au metteur au point. Il permet d'interdire l'accès en écriture à une variable. L'interpréteur se sert de cet indicateur pour empêcher

la modification de la variable de boucle d'une instruction FOR. Nous montrons dans l'exemple qui suit un programme qui produira une erreur d'exécution.

```
PROGRAM essai;
  VAR i : integer;
  PROCEDURE modifie;
    BEGIN
      i := i - 1
    END;
  BEGIN
    FOR i := 1 TO 10 DO
      modifie;
  END.
```

Toutefois, pour donner plus de souplesse au système, il est possible de déverrouiller par une commande du metteur au point cette variable. Ceci est nécessaire pour permettre à l'utilisateur d'exploiter la défaillance des compilateurs lorsqu'aucune autre solution ne s'offre à lui.

Nous aurions pu interdire cela, et considérer que la commande de déverrouillage n'était possible que pour invalider la commande verrouiller et donc interdire la libération de la variable de boucle.

3.3. Gestion de la memoire

La mémoire de l'interpréteur est partagée entre la pile et le tas. Toutes les variables du programme, qu'elles soient globales ou locales à une procédure, sont allouées dans la pile. Les informations allouées dynamiquement par les instructions de gestion dynamique de la mémoire sont allouées dans le tas. Notre gestion de mémoire utilise deux pointeurs qui lui permettent de mémoriser le sommet de la pile et du tas.

Le calcul de l'adresse d'une variable est la somme de

l'adresse de base de la région de déclaration et du déplacement par rapport à cette base. A chaque variable sont associés deux attributs contextuels dans la R.I ; le premier mémorise la région de déclaration et le second le déplacement. A chaque région est associée une adresse de base dans la pile qui est égale à la somme : adresse de base de la région précédente + taille de la région + 1. Cette adresse est donc calculée dynamiquement à l'exécution.

3.4. Traitement des types

Les variables du programmes peuvent être de types divers. Nous allons scinder ces types en trois catégories pour les effets de la présentation.

3.4.1. Les types de base

Pour éviter de typer les cellules mémoires attribuées aux variables du programme interprété, nous implémentons tous les types de base sur un nombre entier de cellules. Cette méthode peu économique en place, (en effet, un booléen, un caractère, et un entier occupent le même espace) est plus efficace du point de vue accès.

Les variables de type réel sont représentées par N cellules de base. Le nombre N est un paramètre du système et dépend de la machine sur laquelle s'exécute l'interpréteur. A chaque type de base, à l'exception des booléens, est associé un ensemble de valeurs dépendant de la machine hôte. Toutefois, cette limite doit être inférieure ou égale à la limite maximum de la machine de développement, pour éviter d'avoir à écrire le code des opérateurs de base (+, -, *, div). Pour les types entiers et réels une limite minimum est également définie.

3.4.2. Les types structurés

Pour les types structurés, il serait possible d'optimiser cette allocation en conservant un seul type de mémoire, et en allouant à une variable structurée un nombre entier de cellules inférieur au nombre d'éléments de base associés à ce type. On pourrait alors mémoriser plusieurs caractères par cellule, ou plusieurs bits dans le cas des types tableau "PACKED", mais ceci nous imposerait une conversion lors de chaque accès.

Les tableaux

Les différents éléments d'un tableau sont rangés dans l'ordre ligne-colonne pour pouvoir réaliser l'instruction d'affectation PASCAL, permettant d'affecter des lignes d'un tableau.

Toutes les informations nécessaires à l'accès à un élément du tableau sont mémorisées dans la déclaration de celui-ci.

L'attribut PACKED du langage est ignoré car nous ne réalisons aucune optimisation dans notre allocation de mémoire.

L'adresse de base d'une variable tableau n'est pas, contrairement à tous les autres types de variables, son adresse réelle. A cette variable est associée une adresse virtuelle (l'adresse de l'élément 0 s'il existait), calculée en fonction des bornes des différentes composantes, ainsi que de leurs enjambées ; ces dernières sont mémorisées dans l'arbre et calculées comme suit :

Soit la déclaration :

```
VAR t1 : array [1..10, 5..14, -5..5] of integer ;
```

La représentation interne associée à cette déclaration est la suivante :

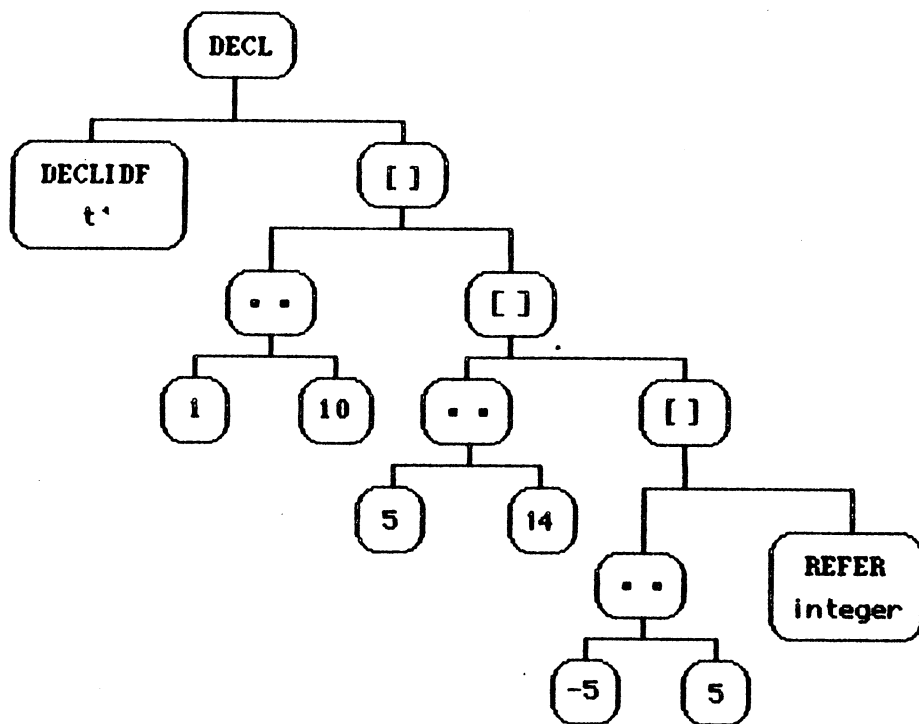


FIGURE 10

La formule utilisée pour calculer l'adresse virtuelle est la suivante :

AV = AR - (N1E1 + N2E2 + ... + NnEn)

où : AR est l'adresse réelle et est égale à :
valeur de la base (AB) + déplacement,
AV l'adresse virtuelle,
N1 .. Nn les bornes inférieures des composantes du tableau.
E1 .. En les différentes enjambées.
En est la taille d'un élément de base du tableau.

La place mémoire occupée par un tableau est :

E1 * abs (Bsup1 - Binf1 + 1)

Après traitement du type, l'arbre correspondant à la déclaration du tableau T1 devient :

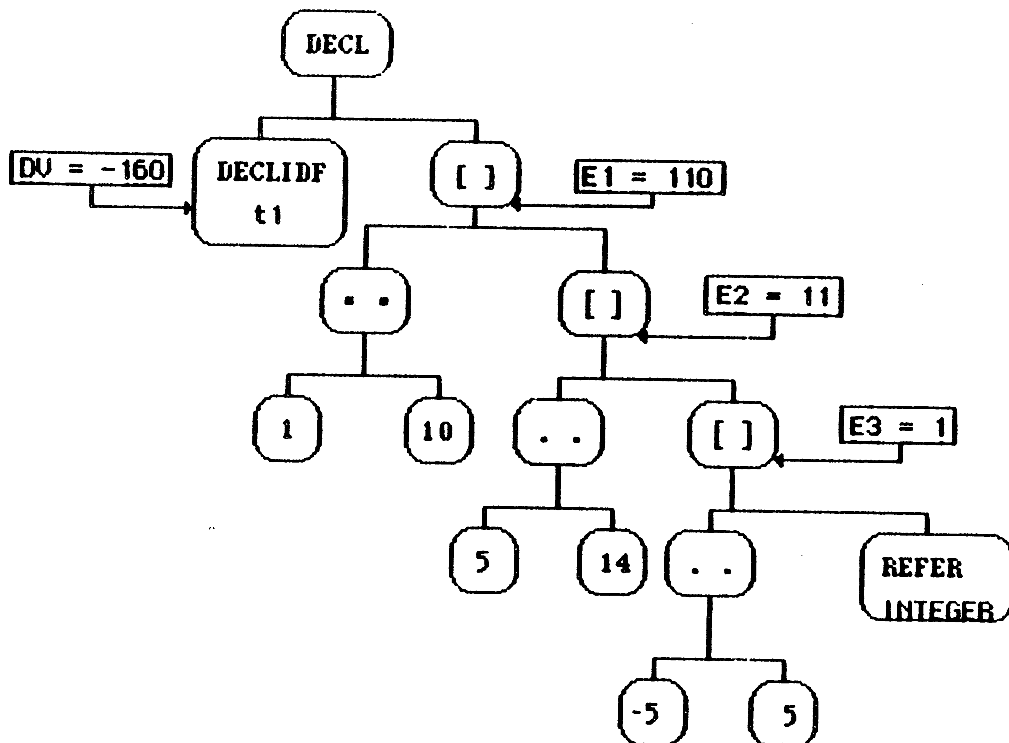


FIGURE 11

L'accès à un élément est alors très simple et rapide. Pour accéder à un élément dont les indices sont : i1, i2, ..., in nous utiliserons l'adresse réelle :

AR = AB + DV + (i1*e1 + i2*e2 + ... + in*en)

où AB = adresse de base du tableau
DV = déplacement virtuel du tableau
i1 .. in = indices de l'élément désigné
e1 .. en = les enjambées mémorisées dans l'arbre.

Les enregistrements

Un type enregistrement PASCAL est constitué par un ensemble de champs fixes, suivi éventuellement d'une partie variante ; chaque champ de la partie variante est à son tour structuré en une partie fixe et une partie variable. Pour sélectionner le champ, le programmeur peut associer à la partie variante un sélecteur de variante. Ce sélecteur est nécessairement de type scalaire et permet de mémoriser la variante active de l'enregistrement à un

instant donné.

La taille d'une variable de type enregistrement est égale à la taille de la partie fixe plus la taille de la variante la plus grande. A la rencontre d'un tel type, l'allocateur décore le sous-arbre associé à la structure par deux attributs. Le premier est placé sur chaque variante et permet de mémoriser la taille de la partie fixe qui précède la variante ; la place occupée par l'éventuel sélecteur de champ est comptée en partie fixe. Le second est associé à chaque identificateur contenu dans cette sous-structure et correspond à son déplacement par rapport au début du champ.

Les ensembles

Tous les types ensembles sont implémentés sur le même espace mémoire par notre allocateur. Nous n'avons pas cherché à optimiser l'espace mais plutôt à accélérer l'exécution des opérations portant sur les ensembles. La taille d'une variable de type ensemble dépend de la machine hôte ; toutefois elle doit permettre d'implanter au minimum l'intervalle de valeurs 0 .. 255. La taille allouée à un ensemble devra donc être telle que :

Nb cellules ensemble * Nb bits cellule >= 256.

Les fichiers

A une variable de type fichier, on alloue une place dans la pile pour mémoriser les différentes informations spécifiques aux fichiers. Pour chaque fichier on réserve :

- seize cellules pour le nom du fichier,
- une cellule pour l'état du fichier,
- une cellule pour l'indicateur fin de fichier,
- une cellule pour l'indicateur fin de ligne,
- une cellule pour la position en nombre de caractères de la fenêtre associée au fichier,
- une cellule pour la position dans le tampon,
- N cellules pour le tampon qui lui est associé.

3.4.3. Le type pointeur

Nous utilisons quatre cellules pour représenter les différents pointeurs. Cette représentation nous permet d'effectuer de nombreux contrôles à l'exécution, et évite bon nombre d'erreurs d'exécutions. En effet, sur les quatre cellules, deux d'entre elles sont utilisées pour chaîner les pointeurs référençant un même objet ; la troisième cellule contient l'adresse dans le tas de l'information pointée ; la quatrième contient l'adresse virtuelle de l'élément référencé si celui-ci est un tableau. Ainsi, après les affectations suivantes :

```
NEW (p1);
p2 := p1;
p3 := p1;
```

on obtient le schéma suivant :

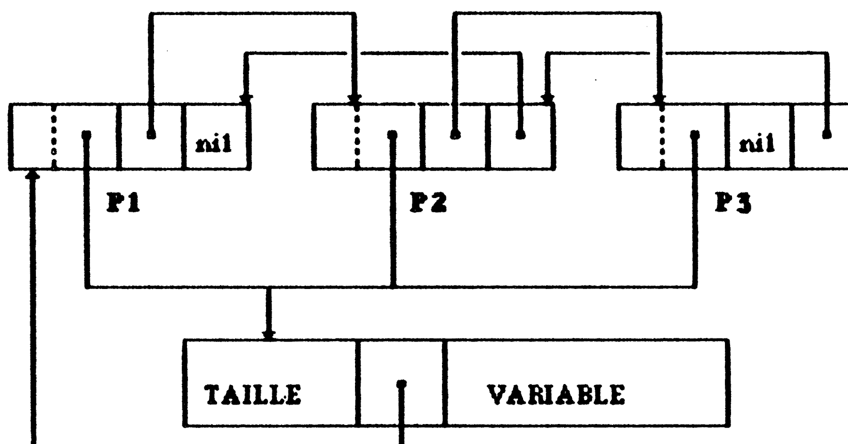


FIGURE 12

Chaque fois qu'un pointeur est affecté par la valeur d'un autre pointeur, il est inséré avant celui-ci dans la chaîne de références correspondante. Le dernier pointeur affecté devient donc la tête de la liste. Ce pointeur hérite par ailleurs des deux adresses contenues dans les champs adresses du pointeur

l'affectant. Cette gestion coûteuse des pointeurs permet :

- de vérifier la libération des variables allouées dynamiquement,
- de ne pas avoir de pointeur référençant des objets libérés.

3.5. Gestion dynamique de la mémoire

Nous avons vu précédement que la mémoire gérée par notre interpréteur était partagée entre la pile et le tas. L'interpréteur alloue ou libère de la mémoire dans deux cas :

- lors d'un appel à une procédure du programme, ne faisant pas partie des procédures standard. Dans ce cas, la mémoire est allouée dans la pile.
- lors de l'interprétation d'une instruction "NEW" ou "DISPOSE". La variable est alors allouée dans le tas.

Deux pointeurs sont utilisés pour réaliser cette gestion de la mémoire : le premier correspond au sommet de pile et le second au sommet du tas. Ces pointeurs croissent en sens inverse ; lorsqu'ils se rencontrent l'interpréteur lance un compactage du tas ("Garbage Collection"). Si la saturation de la mémoire intervient en cours du traitement de l'appel d'une procédure, le ramasse-miettes est lancé automatiquement. Par contre, si cette saturation survient lors du traitement de l'instruction NEW, l'exécution s'arrête et l'utilisateur peut lancer s'il le désire le compactage. Nous décrivons dans ce paragraphe la gestion de la pile ainsi que la gestion du tas.

3.5.1. Gestion de la pile

A chaque appel de procédure est alloué dans la pile l'environnement associé à celle-ci. Le premier élément de cet environnement est une cellule contenant l'adresse de base de l'environnement de cette procédure avant l'appel. Si l'appel correspond à un appel de fonction, l'élément suivant est réservé

pour la valeur à retourner. Sont ensuite réservés le nombre de cellules nécessaires au stockage des paramètres passés par valeur ; suivent ensuite les variables locales de la procédure. Les paramètres passés par adresse héritent des attributs adresse de base et déplacement de l'identificateur correspondant. Lorsque la procédure est désactivée, le pointeur de pile est ramené en début d'environnement. Toutes les cellules allouées sont ainsi libérées et la valeur de la base est restaurée à sa valeur initiale.

3.5.2. Gestion du tas

Chaque information allouée dans le tas peut être schématisé par la figure suivante :

TAILLE GLOBALE DE LA VARIABLE	POINTEUR VERS TETE DE LISTE	MEMOIRE ASSOCIEE A LA VARIABLE

FIGURE 13

La place allouée à une information implantée dans le tas contient, outre les valeurs associées à son type, deux informations supplémentaires : la première, notée Taille sur la figure précédente, indique la taille de l'information allouée y compris ses deux informations complémentaires. Tant que l'information reste active, cette valeur est positive ; lorsqu'elle est libérée, la taille T est alors mise à -T. La seconde, notée Pointeur, mémorise l'adresse dans la pile du premier pointeur de la liste référençant cette variable. Grâce à ces deux informations complémentaires et à la structure des pointeurs, nous pouvons réaliser un ramasse-miettes lorsque la mémoire arrive à saturation.

3.5.3. <u>Ramasse-miettes</u>

La zone d'allocation dynamique de mémoire de l'interpréteur (le tas) est allouée linéairement. Cette gestion peut aboutir à une saturation de cet espace et nécessite par conséquent un ramasse-miettes chargé de récupérer les cellules qui ne sont plus utilisées. L'algorithme de ce ramasse-miettes est très simple ; avant de donner cet algorithme, nous montrons dans la figure 6 comment la pile et le tas sont organisés à un instant donné. Pour donner plus de clarté à ce schéma, nous avons mis la pile et le tas l'un en dessous de l'autre. Cette figure montre que six variables ont été allouées, trois sont toujours actives (V0, V1 et V3) et trois ont été libérées (V2, V4 et V5).

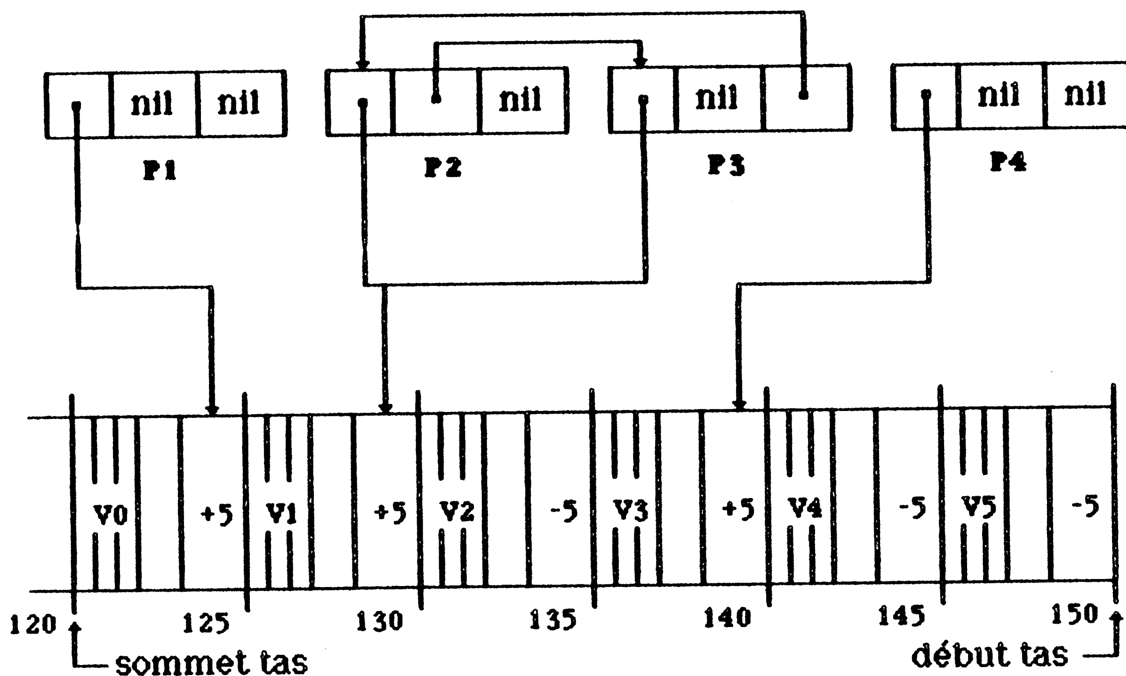


FIGURE 14

L'algorithme du ramasse-miettes est le suivant :

```
DEBUT GC

 - Pointeur_courant := début_tas
       (* Pointeur_courant pointe vers une information du tas *)
 - Nouveau_pointeur := début_tas
 - TANT QUE Pointeur_courant > Sommet_tas FAIRE
   - Longueur_information := Pointeur_courant↑Taille
   - SI Longueur_information > 0
        ALORS - déplacer l'information à l'emplacement
                référencé par Nouveau_pointeur
              - mettre à jour la liste des pointeurs de
                cette information
              - Nouveau_pointeur :=
                     Nouveau_pointeur - Longueur_information
     FINSI
   - Pointeur_courant :=
              Pointeur_courant - abs (Longueur_information)
 - FINTANTQUE
 - Sommet_tas := Nouveau_pointeur
FIN GC
```

3.6. Procédures et fonctions prédéfinies

Les procédures prédéfinies du langage PASCAL permettent de réaliser quatre types de fonctions :

- manipulation de fichiers,
- allocation dynamique de mémoire,
- transfert de données,
- fonctions diverses.

Nous décrivons dans ce qui suit l'implémentation de chacun de ces types de fonctions.

3.6.1. Manipulation de fichiers

Les procédures de manipulation de fichiers imposent une gestion spéciale du tampon alloué à ces fichiers ; en effet, un tampon est vidé ou bien chargé par les procédures WRITELN et READLN. Pour permettre une meilleure mise au point, l'interpréteur gère une fenêtre associée à ce tampon ne correspondant pas à la fenêtre telle qu'elle est définie par le langage. En effet, cette fenêtre est plus large que prévu (un caractère au sens PASCAL). La taille maximum de cette fenêtre est de 75 caractères ; un indicateur associé à ce tampon précise si le buffer a été vidé ou chargé par les ordres respectifs WRITELN ou READLN.
L'interpréteur gère cette fenêtre comme un buffer circulaire. Nous verrons dans la description des commandes de mise au point comment sont exploitées ces informations.

3.6.2. Procédures de gestion dynamique de la mémoire

Le langage PASCAL offre deux procédures prédéfinies (NEW et DISPOSE) pour la gestion dynamique de la mémoire ; ces deux procédures permettent respectivement d'allouer ou de libérer de la mémoire. Elles permettent également de préciser les valeurs qui doivent être données aux sélecteurs de variantes, si l'information concernée est un enregistrement ; cette possibilité est principalement offerte dans un souci d'optimisation de l'espace occupé par ces enregistrements. Dans notre système, cette optimisation n'est pas effectuée car notre objectif est la mise au point du programme et non l'optimisation de celui-ci. Nous allouons donc systématiquement l'espace nécessaire à la variante la plus grande.

Nous avons vu en 3.4.3 que les pointeurs occupent quatre cellules de base. Sur un appel de NEW, nous vérifions si le pointeur est l'élément unique d'une liste de pointeurs. Si c'est le cas nous vérifions alors qu'il ait la valeur NIL. S'il ne l'est pas, un avertissement avec validation est affiché et la variable référencée est libérée. Si le pointeur appartient à une liste de références, il est supprimé de cette liste. Une fois ces

vérifications effectuées, nous allouons la place nécessaire au type référencé par le pointeur.

Sur un appel de DISPOSE, nous effectuons le même type de contrôles. Si le pointeur appartient à une liste de références, nous délions tous les pointeurs de cette liste, nous mettons la valeur de chaque pointeur à nil puis nous rendons négative la taille de l'information allouée. Lors d'une affectation de pointeurs, les vérifications effectuées sur le pointeur affecté sont identiques à celles réalisées sur un appel de NEW.

3.6.3. Procédures de transfert de données

Nous avons vu en 3.4.2 que l'attribut PASCAL "packed" était ignoré. Les procédures PACK et UNPACK sont donc traitées comme des affectations de structures.

3.6.4. Fonctions diverses

Pour les fonctions arithmétiques, l'évaluateur évalue l'expression, vérifie la cohérence du résultat, et si celui-ci est cohérent, lance l'exécution de la fonction. On utilise pour ce faire les fonctions "runtime" système existantes. Pour les fonctions sur les types énumérés (SUCC, PRED et ORD), leur implémentation est très simple, vu l'implémentation de ces types (implantation sur des entiers).

L'interpréteur de notre système est constitué de trois modules principaux ; le premier de ces modules constitue le noyau de cet outil. Le rôle essentiel de ce noyau est de lancer l'exécution des instructions, de gérer les conditions d'arrêt (points d'arrêt, erreur d'exécution, etc.). Le second module constitue l'évaluateur d'expressions du langage. Enfin, le dernier module réalise effectivement les instructions PASCAL.

3.7. Noyau de l'interpreteur

La tâche essentielle du noyau est d'enchaîner l'exécution des instructions du programme traité. Entre chaque instruction, il effectue certaines vérifications et certaines fonctions ; c'est également lui qui gère les points d'arrêt, la temporisation et le pas à pas.

L'interpréteur est lancé avec cinq paramètres qui déterminent le fonctionnement du noyau. Ces paramètres sont les suivants :

Adresse : Cette adresse est celle du noeud sur lequel doit commencer l'interprétation.

Type de lancement : Ce paramètre indique à l'interpréteur s'il doit se réinitialiser avant de continuer. Si c'est le cas, la pile d'exécution est réinitialisée, les différentes variables internes du noyau remises à leur valeur initiale et les variables globales du programme interprété réallouées.

Type d'exécution : Ce paramètre indique le mode d'exécution du programme. Il peut prendre trois valeurs : pas à pas, visuel ou continu. Dans le premier cas, l'interprétation sera arrêtée après l'exécution de chaque pas. Si le mode choisi est le mode visuel, nous affichons après l'exécution du pas la partie du texte source du programme qui contient la prochaine instruction à exécuter ; celle-ci étant mise en évidence, l'exécution se poursuit après une temporisation. Dans le dernier cas, l'exécution se poursuit jusqu'à la rencontre d'une condition d'arrêt.

Temporisation : En mode visuel, cette valeur fixe la durée de

la temporisation observée entre deux instructions. Elle est exprimée en cinquantièmes de seconde et est armée avant l'exécution de chaque pas de programme.

Profondeur : En mode visuel ou en mode pas à pas, ce paramètre indique la profondeur de visualisation correspondant au niveau d'imbrication d'instructions composées.

En fin d'exécution, l'interpréteur retourne deux valeurs qui précisent au module appelant la condition d'arrêt rencontrée (erreur, fin normale, etc.) et l'adresse du noeud sur lequel l'interprétation doit reprendre.

Le noyau gère deux piles durant son exécution. La première est la pile d'exécution qui contient les différents noeuds en cours d'interprétation et qui correspondent à l'imbrication de blocs et d'instructions composées. La fin du programme est détectée lorsque cette pile est vide. Un élément de cette pile est constitué de trois parties :

- l'adresse de l'instruction,
- les indicateurs d'activation,
- l'adresse du dernier noeud évalué par l'évaluateur d'expressions.

La deuxième pile est la pile d'appels des procédures. Chaque élément de cette pile contient :

- l'adresse de l'instruction appelante,
- l'adresse de la procédure appelée,
- la valeur des paramètres à l'appel.

L'interpréteur que nous avons réalisé a une structure itérative. Cette structure lui confère une plus grande simplicité ainsi qu'une plus grande adaptabilité à l'environnement englobant ; elle le rend également indépendant des autres outils et, en particulier, de l'éditeur syntaxique. La gestion des modifications de programme par cet éditeur est réalisée ailleurs,

ce qui rend plus facile la maintenance et l'évolution du système. Toutefois, ce choix implique une réalisation itérative de l'évaluateur d'expressions, ce qui le complique grandement, car la structure arborescente des expressions est mieux adaptée à une programmation récursive. En effet, la présence d'appels de fonctions dans des expressions implique un appel à l'interpréteur et impose donc une évaluation en deux passes. Cet évaluateur est décrit dans le prochain paragraphe.

L'algorithme général de ce noyau est le suivant :

```
DEBUT NOYAU

  -SI type de lancement = initialisation
      ALORS - Initialiser pile_appel
            - Initialiser pile_exécution
            - Allouer_paramètres du programme
            - Allouer_variables du programme
            - Instruction_courante := première instruction
   FINSI
  -TANT QUE non condition_arrêt (instruction_courante) FAIRE
      - exécuter (instruction_courante)
      - instruction_courante := suivant (instruction_courante)
      - SI mode = visuel
          ALORS - afficher (instruction_courante)
                - temporiser
        FINSI
   FINTANTQUE
  - Retourner condition_arrêt (instruction_courante)

FIN NOYAU
```

Les conditions d'arrêt testées avant l'exécution d'une instruction sont, par ordre de priorité :

- l'appel opérateur ("BREAK"),
- une erreur sémantique sur le noeud instruction à évaluer,
- un point d'arrêt sur cette instruction.

L'appel opérateur n'est jamais pris en compte durant l'exécution d'une instruction mais avant d'exécuter l'instruction suivante. Si l'arrêt s'est produit en raison d'une erreur sémantique, celle-ci doit être corrigée avant de relancer l'exécution.

Un point d'arrêt ne provoque pas toujours un arrêt de l'exécution. En effet, on associe à chaque point d'arrêt une valeur indiquant le nombre de fois que l'instruction doit être exécutée avant de provoquer un arrêt effectif.

Après l'exécution d'une instruction, d'autres conditions d'arrêt peuvent se présenter :

- une erreur est provoquée par l'exécution de cette instruction,
- l'exécution est effectuée en mode pas à pas et le niveau de profondeur d'exécution est inférieur à la limite fixée par l'utilisateur,
- la pile d'exécution est vide.

Si l'interpréteur a été lancé en exécution visuelle et si le niveau de profondeur est inférieur à la limite maximum donnée par l'utilisateur, une temporisation est effectuée. Cette temporisation se fait par rapport au temps mémorisé avant le lancement de l'instruction. De ce fait, cette temporisation est plus régulière, car elle tient compte du temps mis pour exécuter le pas précédent.

Le passage à l'instruction suivante est réalisé de manière très simple. En effet, le noyau considère toujours comme instruction courante celle qui est au sommet de la pile d'exécution. Par conséquent, l'enchaînement est effectué en opérant sur cette pile ; ainsi, si l'exécution de l'instruction courante est terminée, c'est elle même qui se désempile ; dans le cas contraire, l'instruction empile la prochaine instruction à évaluer, si elle la connait (instruction conditionnelle, instruction itérative, instruction "begin"). Lorsque la pile est

vide, cela signifie que le programme est terminé.

3.8. Evaluation d'expressions

L'évaluation d'une expression est réalisée en deux temps. En effet, nous évaluons tout d'abord les feuilles du sous-arbre expression. Les résultats sont empilés dans l'ordre des évaluations. Une fois toutes les feuilles évaluées, un deuxième parcours est effectué pour calculer le résultat global. Ceci est nécessaire pour la prise en compte du pas à pas lors de l'appel de fonctions. De cette façon, il est possible d'effectuer toutes les opérations de mise au point dans la fonction. Pour cela, l'évaluateur associe à l'instruction en tête de la pile d'exécution un indicateur qui précise au noyau que l'évaluateur est lancé. D'autre part la feuille terminale évaluée est également associée à cette instruction. L'exemple qui suit illustre le fonctionnement de l'évaluateur. Soit le programme suivant :

```
PROGRAM evaluation ;
  VAR i, j : INTEGER ;
  FUNCTION fonct (k : INTEGER) : INTEGER ;
     BEGIN
       fonct := k + 1
     END ;
  BEGIN
    j := 5 ;
    i := j + fonct (j)
  END.
```

Suivons l'exécution de l'affectation "i := j + fonct (j)" de ce programme. La figure suivante montre l'évolution des différentes piles (pile d'exécution, pile des valeurs, pile des opérateurs) et des indicateurs associés à l'instruction en cours d'interprétation tout au long de l'évaluation.

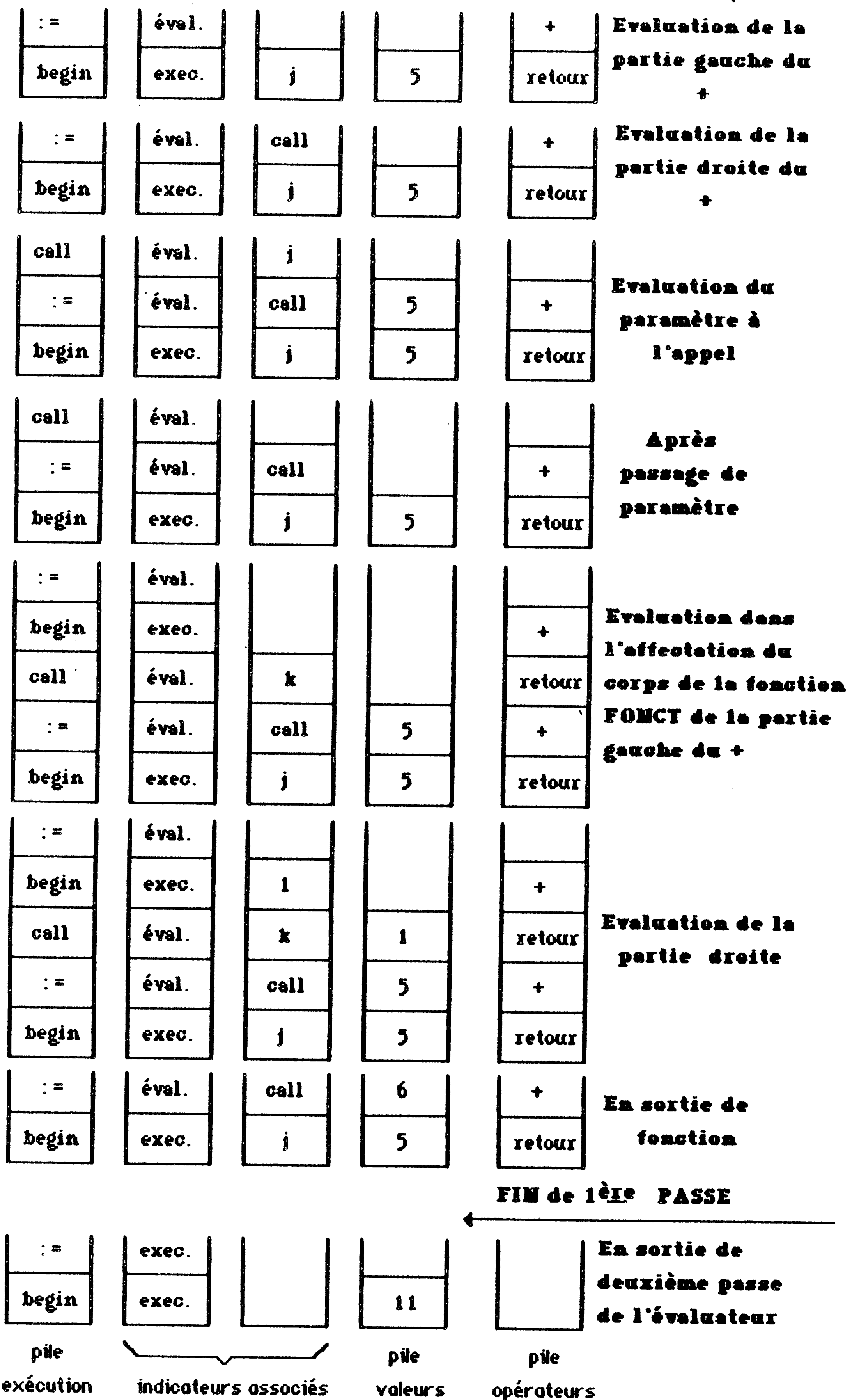


FIGURE 15

3.8.1. Algorithme de l'évaluateur

L'algorithme de l'évaluateur est divisé en deux parties, correspondant aux deux passes décrites précédemment.

```
DEBUT PASSE1

 - TANTQUE (non fin) et (feuille à évaluer) FAIRE
   - SI (type_de_feuille = call) et
        (nom_fonction n'appartient pas aux fonctions prédéfinies)
     ALORS - empiler (feuille_courante)
           - instruction_courante := feuille_courante
           - fin := vraie
     SINON - prendre valeur memoire associée à feuille
           - empiler (valeur_feuille)
           - passer (feuille_suivante)
     FINSI
   FINTANTQUE
 - Retourner (instruction_suivante)

FIN PASSE1

DEBUT PASSE2

 - TANT QUE type opérateur <> retour_valeur FAIRE
    - SI opérateur binaire
        ALORS - désempiler (valeur1)
              - désempiler (valeur2)
              - empiler (calculer (valeur1 valeur2 operateur))
              - désempiler (operateur)
        SINON - désempiler (valeur1)
              - empiler (calculer (valeur1 operateur))
              - désempiler (operateur)
      FINSI
   FINTANTQUE

FIN PASSE2
```

3.9. Traitement des instructions

Ce paragraphe traite de l'implémentation des différentes instructions du langage. Pour chacune d'elles nous décrivons le mécanisme général du fonctionnement, ainsi que l'algorithme utilisé pour sa mise en oeuvre.

3.9.1. Instruction d'affectation

Lors d'une affectation, il est nécessaire de réaliser d'une part des conversions de type et, d'autre part, les contrôles liés aux fonctions de mise au point. L'évaluateur d'expressions rend comme résultat une structure dont le format est le suivant :

- indicateur de type : - élément simple,
  - élément long,
  - tableau,
  - record,
  - erreur.
- valeur : - pour un élément simple, sa valeur,
  - pour un tableau ou record, l'adresse réelle de son premier élément,
  - pour un élément long, cette valeur n'a pas de sens.
- longueur : longueur de l'élément évalué.

Les conversions sont réalisées lorsque l'on affecte un entier à un réel. L'algorithme de l'affectation est le suivant :

DEBUT AFFECTATION

```
- Evaluer partie droite
- SI erreur
   ALORS - retourner (erreur, code_erreur)
   SINON - calculer_adresse (partie_gauche)
         - SI erreur ou protection (partie_gauche)
```

```
        ALORS - retourner (erreur, code_erreur)
        SINON - SI trace (partie_gauche)
                  ALORS - afficher (valeur_partie_gauche)
                        - afficher (valeur_partie_droite)
                FINSI
              - SI conversion
                ALORS - convertir_réel (valeur_partie_droite)
                FINSI
              - affecter (variable, longueur)
              - désempiler (instruction)
    FINSI
FINSI
```

<u>FIN AFFECTATION</u>

Nous pouvons remarquer dans cet algorithme que la trace et la protection des variables sont assurées au sein même du mécanisme d'affectation.

3.9.2. <u>Instruction BEGIN</u>

L'instruction BEGIN est extrêmement simple à interpréter. Si l'élément courant n'est pas le dernier de la liste, elle se contente d'empiler son successeur. L'algorithme est donc le suivant :

<u>DEBUT BEGIN</u>

```
 - incrémenter numéro de fils
 - SI numéro de fils > Nombre_fils
   ALORS - désempiler_instruction     (* BEGIN *)
         - instruction_suivante := nil
   SINON - empiler_instruction (fils (N, instruction_begin))
         - modifier numéro fils en tête de pile
         - instruction_suivante := sommet_pile_exécution
 FINSI
```

<u>FIN BEGIN</u>

3.9.3. Instruction IF

L'instruction de branchement conditionnel fait appel à l'évaluateur pour évaluer sa condition. Si celui-ci retourne une adresse d'instruction (appel à une fonction du programme), l'évaluateur est lancé autant de fois que nécessaire sur l'expression à évaluer. Ceci est valable pour toutes les instructions contenant une expression.

En fin d'évaluation et suivant la valeur retournée, l'une des deux branches de l'instruction est choisie, en fonction de la valeur obtenue.

L'algorithme de cette instruction est le suivant :

```
DEBUT IF

 - TANT QUE valeur_retournée = instruction FAIRE
    - Evaluer condition_if
   FTQ
 - SI valeur = vraie
    ALORS - fils_suivant := partie_then
    SINON - fils_suivant := partie_else
   FINSI
 - désempiler_instruction        (* IF *)
 - SI fils (fils_suivant instruction_courante) <> nil
     ALORS - instruction_suivante :=
                 fils (fils_suivant instruction_courante)
           - empiler_instruction (instruction_suivante)
     SINON - instruction_suivante := nil
   FINSI

FIN IF
```

3.9.4. Instruction CASE

L'instruction CASE fait également appel à l'évaluateur pour évaluer l'expression de sélection. Après retour de sa valeur l'interpréteur parcourt tous les cas jusqu'à ce qu'il trouve le cas correspondant ou la fin de la liste. Dans ce dernier cas, une erreur d'exécution est détectée. L'algorithme de l'instruction CASE est le suivant :

```
DEBUT CASE

 - TANT QUE valeur_retournée = instruction FAIRE
    - Evaluer (expression_case)
   FTQ
 - instruction_suivante = nil
 - TANT QUE non fin
    - SI valeur dans liste_constantes  (cas_courant)
        ALORS - fin := vrai
              - instruction_suivante := instruction (cas_courant)
              - désempiler_instruction
              - empiler (instruction_suivante)
        SINON - SI fin_liste
                 ALORS - fin := vrai
                 SINON - passer au cas suivant
                FINSI
        FINSI
    FINTANTQUE
 - SI instruction_suivante = nil
    ALORS - retourner (erreur)
   FINSI

FIN CASE
```

3.9.5. Instruction WHILE

L'instruction WHILE lance l'interprétation de son corps suivant la valeur retournée par l'évaluation de sa condition. Elle gère un compteur de boucle indiquant le nombre d'itérations réalisées à un moment donné. L'algorithme est donc le suivant :

```
DEBUT WHILE

 - SI premier_passage
    ALORS - empiler_compteur_boucle (0)
   FINSI
 - TANT QUE valeur_retournée = instruction FAIRE
    - Evaluer (condition_while)
   FTQ
 - SI valeur = vrai
    ALORS - incrémenter_compteur_boucle
          - instruction_suivante :=
                      fils (instruction, instruction_courante)
          - empiler_instruction (instruction_suivante)
    SINON - désempiler_compteur_boucle
          - désempiler_instruction
          - instruction_suivante := nil
   FINSI

FIN WHILE
```

3.9.6. Instruction REPEAT

L'interprétation de l'instruction REPEAT est sensiblement la même que l'instruction WHILE. Nous donnons tout de même l'algorithme permettant de traiter cette instruction.

```
DEBUT REPEAT

 - SI premier_passage
    ALORS - empiler_compteur_boucle (1)
```

```
        - instruction_suivante :=
                      fils (instruction instruction_courante)
        - empiler_instruction (instruction_suivante)
   SINON - TANT QUE valeur_retournée = instruction FAIRE
            - evaluer (condition_repeat)
          FTQ
        - SI valeur = vrai
           ALORS - incrémenter_compteur_boucle
                 - instruction_suivante :=
                       fils (instruction, instruction_courante)
                 - empiler_instruction (instruction_suivante)
           SINON - désempiler_compteur_boucle
                 - désempiler_instruction
                 - instruction_suivante := nil
          FINSI
   FINSI
```

FIN REPEAT

3.9.7. Instruction FOR

L'interprétation d'une instruction FOR débute par l'évaluation de ses valeurs initiale et finale ; par ailleurs, la variable de boucle est protégée contre toute modification. Ensuite, l'interpréteur itère sur l'instruction, comparant la variable de boucle et la valeur finale suivant le type du FOR (to ou down) et exécutant le corps de celui-ci jusqu'à ce que cette comparaison donne la valeur "faux". Sur cette instruction, nous utilisons également un compteur de boucle différent de la variable de contrôle, car en cours d'exécution, il peut être impossible de recalculer la valeur initiale.

DEBUT FOR

```
 - SI premier_passage
    ALORS - TANT QUE valeur_retournée = instruction FAIRE
              - évaluer (expression_fin)
```

```
          - évaluer (expression_debut)
          FTQ
        - empiler (valeur_fin)
        - empiler_compteur_boucle (0)
        - protéger (variable_boucle)
  SINON - incrémenter ou décrémenter (variable_boucle)
 FINSI
- SI valeur (variable_boucle) dans les bornes
  ALORS - instruction_suivante :=
              fils (instruction, instruction_courante)
        - empiler (instruction_suivante)
        - incrémenter_compteur_boucle
  SINON - déprotéger (variable_boucle)
        - désempiler_instruction
        - instruction_suivante := nil
        - désempiler (compteur)
 FINSI
```

FIN FOR

3.9.8. Instruction CALL

Nous avons vu dans le chapitre précédent, que les procédures ou fonctions prédéfinies du langage étaient traitées comme des opérateurs. L'algorithme que nous donnons ne s'applique donc qu'aux appels à des fonctions définies dans le programme.

Dans le cas des procédures ou fonctions non prédéfinies, l'appel de procédure lance l'évaluateur d'expressions et l'allocateur de variables. La première cellule allouée dans la pile est l'adresse de l'environ associé à l'appel précédent ; la seconde, si cet appel correspond à un appel de fonction, sert à stocker la valeur de retour de celle-ci ; le nombre de cellules réservées correspond bien entendu à la taille occupée par une variable du type de la valeur retournée. On trouve ensuite dans la pile les paramètres passés par valeur suivis des variables locales. Pour les paramètres passés par variable, l'allocateur copie les attributs adresse de base et déplacement du paramètre

effectif dans les attributs correspondant associés à la déclaration du paramètre.

L'algorithme de cette instruction est le suivant :

DEBUT CALL

```
 - SI premier passage
     ALORS - placer adresse_base de l'environ courant dans pile
           - allouer (paramètres)
           - allouer (variables_locales)
           - placer appel dans pile_appel
           - instruction_courante := première_instruction_begin
           - empiler (instruction_courante)
    SINON - SI type_bloc = fonction
              ALORS - empiler (résultat) dans pile valeur
            FINSI
          - réinitialiser (adresse_base)
          - désempiler_environ
          - désempiler_instruction
          - désempiler_appel
  FINSI
```

FIN CALL

3.9.9. Instruction GOTO

Le GOTO est l'instruction du langage PASCAL la plus difficile à interpréter. Elle est également la plus délicate car il est nécessaire de remettre en état toutes les piles gérées par l'interpréteur : pile d'exécution, pile d'appel, pile de valeurs, pile d'opérateurs, etc. La possibilité de sortir d'un bloc pose en effet de gros problèmes d'exécution. Outre la réinitialisation des piles, il est également nécessaire de fermer les fichiers temporaires qui ont pu être ouverts dans ce bloc ; pour permettre cette fermeture, nous attachons à la déclaration du bloc, un attribut contenant la liste des fichiers temporaires ouverts.

Lorsque l'on parcourt la pile d'exécution pour détecter les appels de procédures, nous désempilons l'environ associé à chaque appel rencontré, en fermant les fichiers correspondants. L'algorithme suivant montre les mécanismes généraux employés, mais ne met pas en évidence le désempilement de toutes les piles gérées.

```
DEBUT GOTO

 - condition_arrêt := père (étiquette)
 - TANT QUE sommet (pile_exécution) <> condition_arrêt FAIRE
   - CAS de sommet (pile_exécution)
     - call : - SI fichiers ouverts
                  ALORS fermer fichiers
                FINSI
              - désempiler_pile_appel
              - réinitialiser (adresse_base)
              - désempiler_environ
     - for :  - désempiler_compteur_boucle
              - désempiler_valeur_fin
     - repeat, while :
              - désempiler_compteur_boucle
     - autres cas : rien
    FINCAS
   - désempiler_instruction
 - FTQ
 - modifier (numéro_fils (sommet (pile_exécution))
 - instruction_suivante := instruction_associée (étiquette)
 - empiler (instruction_suivante)

FIN GOTO
```

3.9.10. Instruction WITH

Cette instruction n'est pas simplement utilisée pour faciliter l'écriture du programme. En effet, la liste d'expression donnée est évaluée à l'entrée du WITH. Leurs valeurs sont ensuite réutilisées partout où les expressions apparaissent dans le corps

du WITH. Cette instruction ne pose pas de problème particulier à l'interprétation. En effet, l'analyseur sémantique crée des liens entre une expression utilisée dans la boucle et la variable réelle. Lors de l'évaluation par l'interpréteur des expressions attachées au WITH, celui-ci stocke ces valeurs dans un attribut spécial attaché à la déclaration de ce noeud. Les problèmes sont donc plus sémantiques qu'interprétatifs.

L'algorithme de cette instruction est le suivant :

```
DEBUT WITH

 -SI premier passage
     ALORS - évaluer (liste_expressions)
           - mémoriser valeurs sur attributs
           - empiler (première_intruction_du_with)
     SINON - supprimer attributs
           - désempiler_instruction
 - FINSI

FIN WITH
```

4. LE METTEUR AU POINT

Dans ce chapitre, nous décrivons les différentes fonctions du metteur au point ADELE, ainsi que l'interface qu'il utilise pour dialoguer avec l'utilisateur. Nous ne donnons pas ici la syntaxe exacte de ces commandes, celle-ci étant donnée en annexe.

4.1. Fonctions

Le metteur au point ADELE permet d'exécuter un programme PASCAL en observant la progression de l'exécution. Ceci est obtenu grâce à une visualisation permanente du texte source du programme et à une mise en relief sur celui-ci de l'instruction courante ; cette visualisation peut par ailleurs être complétée par une trace des variables et des procédures choisies par l'utilisateur.

Il offre également la possibilité de commander l'arrêt de cette exécution sous différentes formes, pour permettre ainsi à l'utilisateur de consulter les différentes informations liées à l'exécution du programme ou de modifier les conditions de celle-ci.

Nous présentons ci-après l'ensemble des fonctionnalités offertes par cet outil. Nous l'illustrons ensuite par un exemple de session de mise au point.

4.1.1. Modes d'exécution

L'exécution d'un programme peut être réalisée de deux manières différentes :

- en mode continu jusqu'à ce qu'une condition d'arrêt quelconque soit rencontrée,
- en mode pas à pas : le metteur au point se met en attente de commande, après l'exécution d'un pas tout en visualisant la prochaine instruction à exécuter.

Exécution continue

Ce mode d'exécution correspond au déroulement traditionnel d'un programme ; tant qu'une condition d'arrêt n'est pas vérifiée, l'exécution est poursuivie. Les différentes conditions d'arrêt sont définies dans ce qui suit.

a) Erreur d'exécution

Les erreurs d'exécution correspondent aux erreurs classiques : débordement de capacité, indice de tableau hors bornes, division par zéro, valeur d'un cas non prévu dans une instruction CASE, etc. En plus de ces erreurs, nous prenons également en compte :

- l'utilisation d'une variable non initialisée,
- la modification d'une variable à accès protégé.

Tous ces cas provoquent un arrêt de l'interprétation du programme sur l'instruction en cours d'évaluation ; tou^efois, aucun effet de bord ne se produit et on peut considérer que l'instruction n'a pas été exécutée.

b) Erreur sémantique

Un programme traité par notre interpréteur peut être incomplet ou même contenir des erreurs contextuelles (variable non déclarée, type d'expression incompatible, etc). Si lors de l'exécution d'un programme, l'interpréteur rencontre l'un de ces cas, il s'arrêtera sur l'instruction correspondante. Le seul moyen de continuer l'exécution est de corriger cette erreur avec l'éditeur, puis de réactiver le programme.

c) Point d'arrêt

Nous avons vu dans la description de l'interpréteur que celui-ci vérifie la présence d'un point d'arrêt avant d'interpréter chaque instruction. Si elle en possède un,

l'interpréteur évalue la valeur qui lui est associée et s'arrête si elle est égale à zéro ; le contrôle est alors donné au metteur au point qui se met en attente de commandes.

d) Appel opérateur

Un appel opérateur ("BREAK") est pris en compte par l'interpréteur après l'exécution complète de l'instruction simple en cours d'évaluation. Ceci nous permet de laisser l'interpréteur dans un état cohérent, en vue d'une poursuite éventuelle de l'exécution.

e) Fin du programme

Il s'agit de la condition d'arrêt normal du programme. L'utilisateur est alors averti que celui-ci s'est bien terminé.

Durant l'exécution, le metteur au point affiche la partie du programme qui contient l'instruction courante et met celle-ci en valeur. En outre, il observe une temporisation après l'exécution de chaque instruction pour permettre à l'utilisateur de se rendre compte de la progression du programme et de l'arrêter par un appel opérateur, lorsqu'il le désire.
Une commande du metteur au point permet de modifier la temporisation entre chaque pas ; nous avons choisi d'exprimer cette temporisation en quarts de seconde, car nous pensons qu'il n'est pas nécessaire de descendre au dessous de ce seuil (quatre instructions sont visualisées en une seconde dans le cas ou le pas correspond à une instruction simple) car le bruit visuel créé par cette fonction rend rapidement pénible la mise au point. Toutefois, il faut pouvoir aller assez vite dans certaines séquences du programme, et donc ne pas trop temporiser en fin d'instruction, pour permettre un pas à pas "automatique" dans les instructions répétitives. Raisonner en quart de seconde peut paraitre gênant, mais l'utilisateur détermine rapidement la valeur des nombres fixant les vitesses minimum et maximum qui lui sont adaptées.

Exécution en pas à pas

L'exécution en mode pas à pas est de deux types. Soit le contrôle est donné à l'utilisateur après l'exécution d'un pas, soit le pas courant est visualisé et l'interprétation continue. Dans le premier cas, il est possible qu'un arrêt de l'exécution survienne si une condition d'arrêt est vérifiée. Si l'exécution est correcte, la commande permettant d'exécuter le pas suivant doit être brève ; la commande doit correspondre à l'enfoncement d'une touche de fonction si le terminal ne possède pas de désignation directe ou, dans le cas des supports de visualisation munis d'un dispositif de "souris", l'utilisation d'une touche de celle-ci.

Dans les deux types d'exécution, l'utilisateur peut fixer la granularité du pas d'exécution. Deux possibilités lui sont offertes :

- Dans la première (exécution composée), le pas est associé à la notion d'instruction PASCAL. Toute instruction est exécutée comme un tout, y compris les instructions composées ("Begin-End", "while", etc.).
- Dans la seconde (exécution simple), le pas est associé à la notion d'instruction simple (affectation, "goto") ; toutes les autres instructions donnent lieu à une décomposition. Il s'agit par conséquent d'une granularité d'exécution plus fine.

En combinaison avec la granularité du pas d'exécution, l'utilisateur à également la possibilité de fixer le niveau de visualisation de son programme. Ce niveau correspond à l'imbrication d'instructions composées. Il est caractérisé par un couple de valeurs, qui détermine le niveau minimum et maximum. Lorsque le niveau courant appartient à l'intervalle fixé par l'utilisateur, la valeur du pas est automatiquement passée à "exécution simple". Ceci permet par conséquent d'observer très finement des fragments de programmes jugés pertinents.

4.1.2. Visualisation du programme

Les fonctions de visualisation du programme ne sont pas à proprement dit des fonctions de mise au point. Elles sont plutôt des fonctions d'intérêt général communes à tous les outils de l'atelier.

L'objectif de ces fonctions est d'offrir à l'utilisateur la possibilité de visualiser son programme de différentes façons et de lui permettre de naviguer dans celui-ci.

En mise au point, la navigation dans un programme est tout à fait différente de la navigation propre à l'édition syntaxique. En effet, durant l'édition, les déplacements sont en général relatifs à l'unité syntaxique courante (père, fils, frère, précédent, suivant, etc) ou bien à une entité syntaxique donnée (partie THEN d'un "IF", condition d'un "WHILE", etc.). Par contre, durant la mise au point on recherche uniquement des séquences d'instructions bien précises du programme et souvent indépendantes du contexte courant. Ce type de désignation correspond donc plutôt à une désignation par numéro de ligne, ainsi qu'à un déplacement par pages, l'équivalent du feuilletage d'un listing.

La numérotation des lignes d'un programme sur un écran peut paraître inutile car l'exécution se fait en mode conversationnel ; toutefois, il faut constater que les supports de visualisation sont, pour l'instant, insuffisants pour ce genre d'application en raison de leur capacité réduite d'affichage. Notre expérience nous a montré que le déplacement par numéro de ligne (ou pseudo numéro) de ligne est très pratique et rapide ; en effet pour dépister des erreurs d'exécution particulièrement difficiles à déceler, il est nécessaire d'avoir une grande partie du programme sous les yeux. Malheureusement, dans l'état actuel des choses, le support papier est le seul à offrir cette possibilité, ce qui implique la présence d'un lien entre la liste et le programme affiché : les numéros de lignes.

Par ailleurs, le décompilateur de notre atelier ne renumérote pas un programme après chaque modification ; il ne le fait que sur demande de l'utilisateur ou au début de chaque session. Le but de

ce choix est de ne pas obliger l'utilisateur à avoir en permanence un listing à jour.

Les fonctions d'affichage que nous offrons sont les suivantes :

- afficher une fenêtre de programme à partir de la ligne N,
- afficher l'écran suivant,
- afficher l'écran précédent.

Le décompilateur est également capable d'éditer la structure d'un bloc PASCAL du programme. Nous voyons sur l'exemple suivant à quoi correspond la structure du programme essai.

```
(*1*) program ESSAI (INPUT, OUTPUT);
   (*2*) const
           (*liste de constante*)
   (*4*) var
           (*liste de variables*)
   (*10*) procedure P1;
           (*corps de la procedure*)
   (*25*) function F1 (I :integer) : integer;
           (*corps de la fonction*)
 (*50*) begin
         (*liste d'instructions*)
 (*200*) end.
```

On peut voir sur cet exemple que le programme ESSAI ne contient ni déclaration de label, ni déclaration de type. Il possède par contre des déclarations de constantes et de variables, ainsi qu'une déclaration de fonction et de procédure. On voit également que le corps de ce programme n'est pas vide. Si on veut visualiser la zone des déclarations de variables, il suffit alors d'exécuter la commande :

    voir_sous-arbre 4

4 étant le numéro de ligne de la zone var.

Par ailleurs, la commande :

```
structure 10
```

aura pour effet d'afficher la structure de la procédure P1, c'est à dire :

```
(*10*) procedure P1;
   (*11*) type
           (*liste de types*)
   (*12*) var
           (*liste de variables*)
   (*14*) begin
           (liste d'instructions*)
   (*24*) end;
```

Enfin, la commande :

```
voir_sous-arbre 12
```

donnera pour sa part le résultat suivant :

```
(*12*) var i,j : integer;
(*13*) b :boolean;
```

Le décompilateur met en évidence les erreurs sémantiques et les sous-arbres incomplets. Pour ces derniers, il remplace l'unité syntaxique manquante par le commentaire adéquat : (*instruction*), (*expression*), etc.
Il met également en évidence les points d'arrêt en plaçant un symbole particulier à côté des numéros de ligne correspondants.

4.1.3. Manipulation de variables

Le metteur au point offre différentes fonctions permettant de manipuler les variables du programme traité. Ces fonctions sont les suivantes :

- voir noms_de_variables,
- modifier noms_de_variables = valeur
- tracer noms_de_variables,
- proteger noms_de_variables.

Il existe également les commandes d'invalidations, correspondant aux deux dernières fonctions : "annuler_trace" et "liberer".

Nous précisons dans ce qui suit le fonctionnement de ces différentes fonctions.

Visualisation de la valeur d'une variable

Cette fonction permet de consulter la valeur d'une variable ou bien la définition d'une constante ou d'un type. S'il s'agit d'une constante, la fonction édite uniquement sa valeur ; si c'est un type, elle édite sa définition PASCAL ainsi que son bloc de définition. Pour une variable, elle fournit sa valeur, accompagnée d'informations complémentaires ; si le programme est actif, elle donne le nom de sa procédure de définition en tenant compte de l'imbrication des blocs ; elle fournit également le type mais aussi, si la variable a été affectée, le numéro de la dernière instruction qui a modifié la valeur de la variable.

Reprenons l'exemple du paragraphe 1.2.
Soit I une variable de la procédure P1, PARAM un paramètre de cette même procédure et CONSTANTE une constante déclarée dans le programme principal.
La commande :

voir I, PARAM, CONSTANTE

déclenche l'affichage suivant :

I : integer (variable de la procédure P1)
I variable non initialisée

PARAM : integer (paramètre par variable de la procédure P1)
PARAM = 5 (affecté dans l'instruction 10)

constante CONSTANTE du programme ESSAI
CONSTANTE = 'a'

Si une variable masque une variable de même nom mais définie dans un bloc englobant, l'accès à cette dernière pourra se faire de deux façons :

- soit en changeant d'environnement (cf. paragraphe 1.5),
- soit en donnant le chemin d'accès de la procédure.

Ainsi par exemple, la procédure P1 du programme ESSAI est active, la commande :
voir I

affiche :

I : integer (variable de la procedure P1)
I variable non initialisée

tandis que la commande

voir ESSAI$I

affiche :

I : integer (variable du programme ESSAI)
I = 20 initialisée au clavier

Si une variable appartient à un bloc inclus dans un ou plusieurs autres blocs, il faut donner tout le chemin d'accès si l'on utilise cette méthode.

Si la variable désignée appartient à un bloc inactif, la fonction se limitera à donner les informations concernant la déclaration de la variable.

voir F1$I affichera

I : integer paramètre par valeur de la fonction F1
La fonction F1 n'est pas active

Tableaux

Dans le cas d'une variable de type tableau, il est possible de visualiser un élément bien précis, mais également plusieurs éléments. Soit la déclaration :

```
    TAB : array [1..5, 1..2, 1..2] of integer,
```

la commande :

```
    voir TAB
```

affichera :

```
    TAB : array [1..5, 1..2, 1..2] of integer
    TAB [1, 1, 1] = 1
    TAB [1, 1, 2] = 2
    TAB [1, 2, 1] = 3
        ....
    TAB [5, 2, 2] = 20
```

Tandis que la commande : voir TAB [, 1, 1] déclenchera l'affichage de :

```
    TAB = array [1..5, 1..2, 1..2] of integer
    TAB [1, 1, 1] = 1
```

```
TAB [2, 1, 1] = 9
TAB [3, 1, 1] = 13
TAB [4, 1, 1] = 17
TAB [5, 1, 1] = 21
```

Il est également possible d'utiliser des variables ou des intervalles à la place d'un indice de tableau. Ainsi, par exemple, si i est une variable entière de valeur 2, la commande :

```
voir TAB [5, 1..i, i]
```

éditera :

```
TAB : array [1..5, 1..2, 1..2] of integer
TAB [5, 1, 2] = 18
TAB [5, 2, 2] = 20
```

Enfin, si plusieurs éléments voisins du tableau ont une valeur identique, ils seront regroupés lors de la visualisation dans un intervalle.

Voici un exemple :

```
TAB1 = array [1..10, 1..5, 1..3] of char
TAB1 [1..3, 1..3, 1..2] = 'a'
TAB1 [3, 3, 3] = 'b'
TAB1 [3, 4..5, 1..3] = 'c'
TAB1 [4..10, 1..5, 1..3] = ' '
```

Enregistrements

La visualisation d'une variable enregistrement est faite en affichant le nom de chaque champ de la structure avec sa valeur. Chaque changement de niveau mis est en évidence en décalant les champs du niveau suivant.

Si l'enregistrement possède une partie variante, nous n'affichons que l'alternative correspondant à la valeur courante du sélecteur. Si toutefois, la valeur du sélecteur est incohérente

ou si celui-ci est inexistant, nous éditons la première alternative de l'enregistrement.

Fichiers

La visualisation d'une variable fichier comprend deux types d'informations ; en effet, elle donne d'une part des renseignements concernant l'état du fichier (liaison externe, ouverture réalisée, etc.) et d'autre part, le contenu du tampon associé au fichier. Ce tampon est édité comme s'il s'agissait d'une chaîne de caractères ; les caractères non imprimables sont remplacés par des caractères spéciaux. Par ailleurs, le prochain caractère devant être lu est mis en évidence sur le support de visualisation (en général, en inverse vidéo). Nous utilisons pour l'édition du tampon une table de correspondance nous permettant de remplacer un caractère donné par une autre séquence de caractères. Nous pouvons ainsi utiliser les caractères semi-graphiques du terminal (s'il en possède), mettre une séquence d'échappement ou bien encadrer un caractère terminal d'une certaine chaîne.

Pointeurs

La visualisation d'une variable de type pointeur n'inclut pas le contenu de la variable. Nous précisons uniquement si le pointeur est initialisé ou non et si d'autres pointeurs pointent vers la même adresse.

Pour une variable de type énuméré, nous donnons la valeur symbolique de cette variable ainsi que le rang de cette valeur.
Les variables de type ensemble sont éditées par une suite de "un" et de "zéro".

Modification de variables

Il est possible de modifier le contenu d'une variable lorsque celle-ci est active. Le type de la valeur doit être compatible avec celui de la variable. Pour les variables de type réel, nous

n'assurons pas à ce niveau la compatibilité de type faisant que les entiers sont compatibles avec les réels. Si l'utilisateur donne une seule valeur pour modifier une variable structurée, toutes les composantes seront initialisées à la même valeur.

Exemples :

```
  changer I = 5

  I : integer (variable du programme PROGRAMME)
  I ( 3 ) <--- 5

voir I

  I : integer (variable du programme PROGRAMME)
  I = 5 (valeur affectée au clavier)

changer TAB = 0
  TAB : array [1..10, 1..5, 1..3] of integer
        (variable du programme PROGRAMME)
  TAB [1..10, 1..5, 1..3] <--- 0
```

<u>Trace de l'évolution d'une variable</u>

Cette fonction permet de suivre dynamiquement l'évolution des variables du programme. En effet, si la trace d'une variable est demandée, toute modification de celle-ci sera signalée à l'utilisateur dans une fenêtre spécialisée. Nous précisons alors l'ancienne valeur de la variable, la valeur se substituant à cette dernière ainsi que le numéro de l'instruction qui a modifié la valeur. Nous indiquons au paragraphe 3.2 comment est organisée cette fenêtre de visualisation.

Cette fonction est d'une grande utilité, spécialement pour suivre l'évolution des variables d'état d'un programme et pour dépister les erreurs les plus difficiles (mauvaise initialisation d'une variable, par exemple).

Dans le cas d'un identificateur de fonction ou de procédure, nous éditons à l'appel du bloc l'instruction appelante et la valeur de chaque paramètre lors de leur évaluation. Au retour, nous affichons la valeur des paramètres et, dans le cas des fonctions, la valeur retournée.

Protection d'une variable

Le metteur au point offre la possibilité de protéger une variable contre toute modification. Une tentative de modification d'une variable protégée entraîne l'arrêt de l'exécution du programme ; dans ce cas, la variable doit être "déprotégée" pour pouvoir continuer l'exécution. L'instruction "FOR" utilise cet indicateur de protection pour marquer la variable de boucle. La commande "libérer" permet de passer outre ce contrôle.

Après avoir décrit les différentes fonctions de manipulation d'identificateur, nous allons maintenant passer en revue les commandes de manipulation d'un autre type d'objet : les points d'arrêt.

4.1.4. Manipulation de points d'arrêt

Des points d'arrêt peuvent être attachés aux instructions du programme ; ils permettent d'interrompre l'exécution avant ou après une instruction marquée par un tel objet. Les points d'arrêt de notre système sont des objets valués ; en effet, une valeur entière peut être associée à chacun d'eux pour indiquer que le point d'arrêt ne doit être effectif que lors de la Nième exécution de l'instruction (N étant la valeur associée au point d'arrêt). Cette possibilité est surtout utilisable et n'a de sens réel qu'à l'intérieur des instructions itératives telles que : WHILE, REPEAT ou FOR.

Pour positionner un point d'arrêt, il suffit de donner le numéro de l'instruction et éventuellement la valeur à associer à cet objet.

Exemples :

```
arret-avant 10
arret-apres 11 3
```

Dans le premier cas, le contrôle sera rendu à l'utilisateur lorsque l'interpréteur arrivera pour la première fois sur l'instruction 10, sans l'exécuter. Dans le second cas, l'interpréteur ne s'arrêtera qu'après la troisième exécution de l'instruction 11 ; le pointeur d'instruction restera toutefois sur cette instruction.

L'utilisation de la commande arrêt-après est nécessaire pour arrêter l'exécution d'une instruction itérative à la fin de chaque boucle. L'arrêt après est également utile pour s'arrêter en fin d'exécution sans sortir du programme.

La commande voir-arret [nom procedure] [adresse]

permet de visualiser les différents points d'arrêt d'un programme ainsi que leur valeur courante. Les deux paramètres sont optionnels et permettent, lorsqu'ils sont présents, de restreindre le champ d'action de la commande. On peut ainsi limiter la visualisation des points d'arrêt à une procédure ou même à une instruction donnée. La commande de suppression des points d'arrêt a les mêmes paramètres que cette fonction ; toutefois, lorsque aucun paramètre n'est fourni, une confirmation sera demandée à l'utilisateur.

Les points d'arrêt sont mis en évidence dans le texte source par le décompilateur. Un signe spécial (caractère semi-graphique du terminal) indique la présence du point d'arrêt ainsi que son type (avant ou après). Toutefois, la valeur qui lui est associée n'apparait pas sur le texte source.

4.1.5. Accès aux informations liées à l'exécution

Notre système offre à l'utilisateur la possibilité de manipuler certaines variables de l'interpréteur, liées à l'exécution du programme utilisateur. Ces variables sont :

- la pile d'appel,
- le pointeur environnement courant (ou procédure courante),
- les compteurs de boucle.

Enfin pour permettre une meilleure adaptabilité de l'interpréteur aux compilateurs, il est nécessaire d'offrir à l'utilisateur la possibilité d'inhiber le contrôle de certaines erreurs d'exécution, ainsi que de préciser la façon dont sera fait le "RESET" implicite sur le fichier "INPUT".

La pile d'appel

La pile d'appel des procédures ou fonctions est constituée d'informations reflétant indirectement l'état de la mémoire, et directement le chainage d'appel des procédures. Chaque élément de cette pile est constitué des quatre informations suivantes :

- le niveau d'appel,
- le nom du bloc activé,
- la valeur des différents paramètres au moment de l'appel,
- l'adresse de l'instruction où a été fait l'appel.

a) Le niveau d'appel

Il s'agit d'un entier précisant le rang de l'élément dans la pile. Le niveau le plus élevé est automatiquement associé au pointeur d'environnement après chaque exécution.

b) Le nom du bloc activé

C'est le nom de la procédure ou de la fonction dont

l'appel a provoqué la création de cet élément de la pile.

c) La liste des paramètres

C'est la liste de valeurs associées aux paramètres de la procédure ; les noms de ces paramètres ne font pas partie de cette liste.

d) L'adresse de l'instruction d'appel

Cette adresse sert à mémoriser le numéro de l'instruction ayant effectué l'appel de la procédure ; c'est une référence au texte source.

Une commande permet de consulter la pile d'appel :

voir-pile [niveaux]

Le paramètre permet d'indiquer le nombre de niveaux que l'on veut visualiser, à partir du sommet de la pile ; ce paramètre est optionnel. Cette fonction met en évidence le pointeur d'environneme.t en face de l'instruction empilée ; pour mettre en évidence cette instruction, nous utilisons un marqueur et l'effet inverse vidéo, si le terminal le possède.

<u>Le pointeur d'environnement</u>

Nous avons présenté dans les paragraphes précédents la technique de désignation absolue d'une variable, consistant à donner le chemin d'accès complet à la variable spécifiée (nom de la variable précédé par la liste de ses procédures englobantes). Cette technique a deux inconvénients importants :

- elle ne permet pas de désigner les variables masquées par les appels récursifs d'une procédure,
- elle devient rapidement fastidieuse quand le nombre de désignations augmente.

Le pointeur d'environnement nous permet d'utiliser une deuxième

technique de désignation. En effet, cette variable désigne en permanence l'environnement d'une procédure donnée ("procédure courante"), c'est à dire le segment de données correspondant à celle-ci. Elle permet donc de désigner directement tout objet appartenant à la procédure courante. Par ailleurs, le pointeur d'environnement peut être déplacé sur n'importe quel élément de la pile d'appel, permettant ainsi une désignation plus simple mais surtout l'accès à toutes les instances d'une procédure appelée récursivement. Le pointeur d'environnement est affecté à la procédure sommet de pile à l'activation de chaque procédure.

La commande permettant de déplacer ce pointeur est la suivante :

placer-environnement <niveau>

Le paramètre indique le niveau de la pile sur lequel le pointeur doit être placé. Cette commande affiche en sortie trois élements de la pile : l'élément de niveau supérieur, l'élement désigné, et l'élement de niveau inférieur. Lorsque l'exécution est relancée, le pointeur est réinitialisé à la valeur du sommet de pile.

Une autre fonction de manipulation de ce pointeur permet de désempiler réellement un certain nombre d'appels de procédure et de relancer l'exécution à l'endroit désiré (instruction d'appel de bloc) en réinitialisant les variables globales du bloc le plus interne à leurs valeurs au moment de l'appel. Cette fonction permet ainsi de réaliser une "pseudo-marche arrière" dans un programme.

Les compteurs de boucle

Les compteurs de boucle sont des attributs associés par l'interpréteur aux instructions répétitives (FOR, WHILE, REPEAT). Ces compteurs sont incrémentés à chaque itération sur l'instruction, et réinitialisés en sortie ; Ils sont par ailleurs empilés à chaque entrée de boucle pour tenir compte d'un éventuel

appel récursif. L'intérêt de ces compteurs est très grand car il permet de connaitre le numéro d'itération à tout moment et, en particulier de déterminer s'il y a eu une erreur d'exécution dans une boucle. La commande suivante permet de visualiser ces compteurs :

voir-compteur <instruction>

Modification des contrôles d'exécution

Quatre commandes sont fournies pour manipuler les contrôles effectués lors de l'interprétation du programme. Ces commandes sont les suivantes :

- contrôles-initiaux,
- voir-contrôles [entier],
- supprimer-contrôle <entier>
- valider-contrôle <entier>

Lorsque la commande "voir-contrôles" est sans paramètres, la liste des contrôles est affichée. Pour chaque contrôle d'exécution nous affichons trois informations :

- le numéro d'identification,
- le message associé au contrôle,
- l'état du contrôle.

Le premier champ permet à l'utilisateur de référencer le contrôle désiré. C'est le numéro utilisé dans les commandes "voir-contrôle", "supprimer-contrôle" et "valider-contrôle". Le deuxième est le message édité lors de la détection d'une erreur lorsque l'utilisateur est en mode novice. L'état d'un contrôle peut être soit "effectif", soit "ineffectif".
Nous avons également placé dans la liste de ces contrôles l'indicateur de lecture anticipée. Cet indicateur permet de spécifier le type de lecture que l'utilisateur veut réaliser sur l'entrée standard "INPUT". Il prendra pour valeur soit "anticipée"

soit "à la demande". C'est la valeur "anticipée" qui correspond à l'état "effectif" des autres contrôles.
La commande "supprimer-contrôle" affecte au contrôle désigné la valeur inverse de la valeur initiale. A l'inverse, la commande "valider-contrôle" réalloue la valeur initiale au contrôle désigné.
Enfin, la commande contrôles-initiaux réinitialise tous les contrôles à leur état initial, c'est à dire aux valeurs mémorisées dans le profil de l'utilisateur.

Nous allons maintenant décrire la façon de mémoriser un ensemble de commandes.

4.2. Les contextes d'exécution.

Les contextes d'exécution sont des fichiers de commandes de mise au point, exécutables en bloc par appel d'une commande appropriée. Deux types de commandes sont offertes pour ce faire : un premier type permettant d'éditer un contexte, le second permettant d'activer, de désactiver ou d'exécuter un contexte.

Les commandes offertes pour l'édition des contextes d'exécution sont rudimentaires mais suffisantes pour l'application. Elles permettent d'insérer ou supprimer une commande dans un contexte donné, de créer ou détruire tout un contexte. Les commandes sont stockées dans un fichier sous une forme codée via l'analyseur de commandes de l'environnement. Le fait de passer par cet analyseur procure trois avantages à notre système :

- la commande stockée est toujours correcte,
- on peut stocker des commandes rentrées à l'aide d'un menu,
- on n'analyse la commande qu'une seule fois.

Un contexte est exécuté à la demande explicite de l'utilisateur. Toutefois, il peut également être utilisé comme un prologue, chaque fois que le contrôle sera donné à l'utilisateur.

La commande :

use <nom contexte>

permet d'indiquer au metteur au point qu'un contexte donné doit être utilisé comme prologue.
La commande "fin-contexte" invalide la commande précédente.

La plus grande utilité de ces contextes est l'initialisation du metteur au point : validation ou invalidation de contrôles d'exécution, mémorisation de points d'arrêt pendant une période de tests, groupe de commandes souvent exécutées, etc.

Nous allons maintenant montrer l'utilisation de ces commandes au travers de l'exemple suivant.

4.3. Exemple d'une session de mise au point

Avant de donner un scénario de mise au point, nous précisons certaines conventions adoptées ici. Tout d'abord, le caractère "?" en tête de ligne précise que le metteur au point est en attente de commande. Les caractères "=>" indiquent que l'interpréteur est en attente de lecture. Les lignes encadrées de "***" constituent les messages d'erreurs édités par l'interpréteur lorsque l'utilisateur est en mode novice. Les messages encadrés par "---" correspondent à l'édition des traces. A ceci, nous avons rajouté des commentaires pour faciliter la compréhension de la session et éviter ainsi des références arrières vers la présentation des fonctionnalités. Ces commentaires ont la structure des commentaires PASCAL : (* commentaire *).
Les numéros de ligne apparaissant dans le programme et constituant également des commentaires sont rajoutés par le décompilateur du système.

4.3.1. Programme à mettre au point

```
(* 1*) program SESSION (INPUT, OUTPUT) ;
 (* 2*) var I, J, N : integer ;
 (* 3*) function FACT (VALEUR : integer) : integer ;
  (* 4*) begin
   (* 5*) I := 5 ;
   (* 6*) if VALEUR <= 1
            then
    (* 7*) FACT := 1
            else
    (* 8*) FACT := VALEUR * FACT (VALEUR - 1)
          end ;
 (* 9*) begin
  (*10*) writeln ('Combien de factorielles ',
                   'voulez-vous calculer ?') ;
  (*11*) for I := 1 to N do
   (*12*) begin
    (*13*) writeln ('Donnez le nombre dont vous desirez ',
                     'calculer la factorielle') ;
    (*14*) read (J) ;
    (*15*) write ('La factorielle de ', J : 4, 'vaut : ') ;
    (*16*) j := fact (J) ;
    (*17*) writeln (J : 4)
            end
          end.
```

4.3.2. Session de mise au point

```
? run (* execution du programme sur la racine de l'arbre *)
 =>
(* attente en lecture par l'interpreteur pour lecture anticipee *)
(* on tape RETURN et l'execution se poursuit *)

 Combien de factorielles voulez-vous calculer ?

 *** erreur d'execution dans l'instruction 11   ***
 *** variable non initialisee dans l'expression ***

? voir N  (* visualisation du contenu de la variable N *)
 N : integer (variable du programme SESSION)
 N variable non initialisee

? changer N = 5  (* modification de la valeur de N *)
 N : integer (variable du programme SESSION)
 N (non initialisee) <--- 5

? arret-avant 17 1
 (* positionnement d'un point d'arret sur l'instruction 17 *)
 OK
? tracer FACT  (* on veut tracer la fonction FACT *)
 OK
? arret-avant 7
 (* positionnement d'un point d'arret sur l'instruction 7 *)
 OK
? go  (* on relancer l'execution sur le point courant *)
 Donnez le nombre dont vous desirez calculer la factorielle ?
 => 4 3 5 2 1

 (* on a entre au clavier plusieurs valeurs *)
 (* pour remplir le buffer d'entree         *)
 (* L'execution se poursuit et il apparait  *)
 (* la trace de l'appel de FACT             *)
```

```
--- instruction 16 : FACT (4) ---

(* l'execution est stoppee par une erreur *)

*** erreur d'execution dans l'instruction 5 ***
*** modification d'une variable protegee    ***

? voir I  (* on consulte la valeur de I *)
 I : integer (variable du programme SESSION)
 I = 1 initialisee dans l'instruction 11

? voir-compteur 11

(* on consulte la valeur du compteur de boucle associe a *)
(* l'instruction iterative FOR pour connaitre le nombre  *)
(* d'iteration deja effectuee                            *)

 1 iteration sur l'instruction (*11*) for I := 1 to N do

? liberer I
  (* la variable I a ete protegee en ecriture      *)
  (* par l'instruction FOR du programme principal *)
 OK
? voir INPUT OUTPUT VALEUR
 INPUT : text (parametre du programme SESSION)
 4 /3/ 5 2 1

 OUTPUT : text (parametre du programme SESSION)
 La factorielle de    4 vaut :

 VALEUR : integer (parametre par valeur de la fonction FACT)
 VALEUR = 4 affectee dans l'instruction 16

? go
  (* on relance l'execution, puis apparait la *)
  (* trace des appels recursifs a FACT        *)
```

```
 --- instruction 8 : FACT (3) ---
 --- instruction 8 : FACT (2) ---
 --- instruction 8 : FACT (1) ---
?
 (* arret sur le point d'arret place en 7 *)
? voir-pile  (* on consulte l'etat de la pile des appels *)

 => 4  FACT (1)  <-- 8
    3  FACT (2)  <-- 8
    2  FACT (3)  <-- 8
    1  FACT (4)  <-- 16
    0  SESSION
? voir VALEUR
 VALEUR : integer (parametre par valeur de la fonction FACT)
 VALEUR = 1 affectee dans l'instruction 8
? placer-environnement 2
  (* on place le pointeur de pile sur le deuxieme appel de FACT *)
 OK
? voir-pile
    4  FACT (1) <-- 8
    3  FACT (2) <-- 8
 => 2  FACT (3) <-- 8
    1  FACT (3) <-- 16
    0  SESSION
? voir VALEUR
  (* on consultė la valeur de VALEUR associee *)
  (* au deuxieme appel de FACT                *)

 VALEUR : integer (parametre de la fonction FACT)
 VALEUR = 3 affectee dans l'instruction 8
? voir-arret (* on demande la liste des points d'arrets *)
 instruction 17 = 1    instruction 7 = 1
? voir-arret FACT
  (* on restreint cette fonction a la fonction FACT *)
 instruction 7 = 1
```

```
? supprimer-arret 7
 (* on supprime le point d'arret sur l'instruction 7 *)
 OK
? go
  (* resultat de la trace en sortie de fonction *)
 --- instruction 8 : FACT (1) retourne 1 ---
 --- instruction 8 : FACT (2) retourne 2 ---
 --- instruction 8 : FACT (3) retourne 6 ---
 --- instruction 8 : FACT (4) retourne 24 ---
 (* arret avant l'instruction 17 *)
? detracer FACT  (* on supprime la trace de FACT *)
 OK
? evaluer 16  (* on evalue en local FACT (4) *)
 J := FACT (j) ;
 expression = 24
? go
 La factorielle de    4 vaut :   24

 (* l'execution s'est bien terminee, l'interpreteur s'arrete *)
 *** fin normale d'execution ***
```

5. ASPECTS INTERACTIFS

Dans ce paragraphe, nous décrivons l'interface utilisateur utilisée par le metteur au point. Nous retraçons ensuite les expériences que nous avons menées sur différents types de terminaux pour la mise en oeuvre de cette interface.

5.1. Dialogue metteur au point utilisateur

Dans cette partie, nous nous limitons à présenter les besoins spécifiques du metteur au point en ce qui concerne l'interface utilisateur.

5.1.1. Informations à visualiser

La mise au point de programmes nécessite des besoins bien particuliers. En effet, les objets à afficher ou à mettre en valeur simultanément sont de types différents et ils peuvent ne pas pouvoir cohabiter. Ceci implique par conséquent un multifenêtrage. Les objets à visualiser sont les suivants :

- le texte du programme,
- la structure du programme,
- les points d'arrêts et le pointeur d'interprétation,
- les erreurs sémantiques du programme,
- les données imprimées par le programme interprété,
- les traces de variables et les erreurs d'exécution.

Durant l'exécution, il faudrait par conséquent avoir :

- une fenêtre contenant la partie du programme en cours d'interprétation,
- une fenêtre pour afficher les entrées/sorties de ce programme,
- une fenêtre pour afficher les traces d'exécution,
- une fenêtre pour le dialogue metteur au point-utilisateur.

Toutefois, suivant le type de terminal utilisé, il est possible de regrouper dans une même fenêtre :

- les commandes de mise au point avec les entrées/sorties du programme et les traces d'exécution,
- la structure du programme avec le texte source.

La visualisation d'un objet de type "variable" doit donner à l'utilisateur le maximum de renseignements sur cet objet, sans avoir à exécuter d'autres commandes pour avoir des renseignements supplémentaires. Toutefois, ces informations ne doivent pas surcharger l'espace d'affichage, en fournissant des informations inutiles dans un contexte donné.

Rappelons les renseignements à éditer sur ce type d'objet :

- le bloc de déclaration de la variable, qui permet à l'utilisateur de vérifier qu'il s'agit réellement de la variable demandée.
- son type qui est très utile pour des variables structurées,
- sa valeur,
- l'instruction ayant modifiée cette variable pour la dernière fois.

En ce qui concerne cette dernière information, nous pensons qu'un numéro de ligne est suffisant, surtout si la mise en valeur de cette instruction est fournie. En effet, la majorité des instructions modifiant la valeur d'une variable sont des instructions d'affectation. De ce fait, donner uniquement le texte source de cette instruction peut être peu éclairant, car il est nécessaire de voir le contexte dans lequel est placé cette instruction. D'autre part, afficher la partie de programme contenant cette instruction peut également être fatiguant pour l'utilisateur. Par contre, un numéro de ligne reste un élément important, compte tenu des supports de visualisation existants. Ce numéro se justifie également lorsque l'utilisateur demande de tracer une variable. Deux solutions sont alors possibles : soit afficher uniquement l'ancienne et la nouvelle valeur de la

variable, soit imprimer en plus l'instruction dans laquelle s'effectue la modification. Dans ce dernier cas, il est impossible de changer de fenêtre pour mettre en évidence cette instruction.

Les points d'arrêt et le pointeur d'interprétation sont visualisés dans la fenêtre contenant le texte source. Les points d'arrêt sont mis en évidence par un curseur spécial placé avant ou après l'instruction, suivant le type du point d'arrêt.
La mise en évidence du pointeur d'interprétation peut se faire de différentes façons. La première manière de le réaliser est d'utiliser un autre caractère spécial. Ceci permet de désigner une seule ligne de programme, même si l'instruction est une instruction composée et visualisée sur N lignes. Cette solution offre trois avantages : limiter au maximum le bruit visuel engendré par l'exécution du programme, faciliter le suivi de l'exécution et être simple à implémenter.
Cependant, pour les puristes, il est possible de mettre en évidence une instruction complète (instruction composée). Pour cela, deux solutions sont envisageables. On peut placer toute l'instruction en inverse-vidéo ou en surbrillance. L'exécution d'un pas placera alors le pointeur d'interprétation sur une instruction du bloc. Il faut alors supprimer l'effet vidéo associé à l'instruction englobante et l'affecter à l'instruction courante. On constate très vite qu'une telle solution rend pénible la mise au point en mode pas à pas. La dernière solution est un compromis entre les deux citées précédemment. Le curseur (caractère discret : exemple trait vertical, deux points, barre ouverte, etc.) est placé en marge de toutes les lignes visualisant l'instruction courante et constitue ainsi un trait vertical. Au fur et à mesure que l'exécution continue dans le corps de l'instruction, le trait ainsi affiché diminue en hauteur. Toute l'instruction est ainsi mise en évidence avec un bruit visuel restreint. L'ajustement de la taille du pointeur (longueur du trait) est aussi rapidement réalisable. Le seul inconvénient de cette méthode est le manque de netteté sur une instruction simple, c'est pourquoi nous avons choisi la première solution.
Le seul effet vidéo nécessaire est utilisé pour les erreurs

contextuelles. (surbrillance). Les messages d'erreurs sont assez simples et bref.

Lors d'une erreur d'exécution, nous affichons dans la fenêtre correspondant au texte source la partie du programme contenant l'instruction qui a provoqué l'erreur, celle-ci étant indiquée par le pointeur d'exécution. Puis, nous affichons dans la fenêtre de dialogue le message d'erreur, suivi des éléments ayant déclenché l'erreur : l'expression correspondante et le résultat de son évaluation, le nom du tableau suivi des différents indices, etc.

Les traces de variables peuvent être mises en oeuvre de deux façons différentes, en fonction du support de visualisation. Si la surface de l'écran n'est pas suffisante les traces sont éditées dans le flux des entrées/sorties du programme qui s'exécute. Par contre, s'il est possible d'allouer à cette fonction une fenêtre spéciale, nous le faisons en l'organisant comme suit :

La fenêtre étant composée de N lignes, il est possible d'attribuer une place fixe dans cette fenêtre à N-1 variables ; cette place est mémorisée dans un attribut rajouté à la R.I du programme. La dernière ligne est réservée aux autres variables.

Cette solution permet d'une part de ne pas limiter le nombre de variables à tracer, et d'autre part d'avoir en permanence sous les yeux la valeur des variables auxquelles on attache le plus d'importance, tout en allouant une autre ligne aux autres traces.

5.1.2. Informations à désigner

En mise au point, seul l'accès global aux instructions et aux variables est nécessaire. Ceci permet de n'utiliser qu'un numéro de ligne ou d'instruction pour réaliser la mise au point.

De plus, contrairement à l'édition syntaxique, les commandes renvoient en résultat des références vers des parties de programme ne faisant pas partie du fragment de programme incluant le point courant (référence à l'instruction ayant modifié la valeur d'une variable, déclaration d'une variable, adresse dans la pile d'appel dans la pile des appels, etc.).

5.2. Expériences réalisées

Nous avons mené trois expériences différentes d'interface utilisateur. La première expérience utilise un terminal de 24 lignes de 80 caractères chacune ; la deuxième à été faite sur deux écrans interconnectés. Dans les deux cas, les terminaux possèdaient une gestion rudimentaire de fenêtres (fenêtre uniquement horizontale), des effets spéciaux du type clignotement, inverse-vidéo, surbrillance, des caractères semi-graphiques, et la possibilité d'adresser le curseur. La troisième expérience à été effectuée sur un matériel possédant un écran de type "bit-map" de 1024 X 768 points connecté à une "souris" et permettant l'utilisation de menus.
Dans les paragraphes qui suivent, nous présentons ces trois expériences et les résultats obtenus.

5.2.1. Expérience sur un écran alphanumérique

Nous utilisons pour cette expérience un terminal VT100 comportant l'option semi-graphique. L'écran de ce terminal est découpé en deux fenêtres. La première est assignée à la visualisation du texte programme, sous quelque forme que ce soit : structure ou texte source complet. La seconde est réservée au dialogue avec l'utilisateur ainsi qu'aux entrées/sorties du programme exécuté.

Nous pouvons tirer deux conclusions cette expérience :

a) La surface disponible pour le texte source est très nettement insuffisante pour ce genre d'application. En effet, la mise au point d'un programme nécessite une vision d'une partie importante du texte source : nous considérons qu'il faut au minimum une trentaine de lignes (une demi-page de liste) pour pouvoir être à l'aise. De plus, il est nécessaire de pouvoir changer rapidement de région dans le programme, d'où le besoin d'une liaison rapide entre le calculateur et le terminal. Par contre,la désignation n'est pas un problème majeur.

b) Le fait d'avoir une petite surface d'affichage oblige l'utilisateur à travailler systématiquement avec une liste à côté de lui. Ceci n'est pas très gênant car les numéros de ligne affichés sur l'écran établissent la correspondance entre le texte affiché et la liste du programme (rappelons que l'éditeur syntaxique ne modifie ces numéros de lignes qu'à la demande de l'utilisateur). Ce qui est plus gênant, c'est le multiplexage dans la deuxième fenêtre des entrées/sorties du programme, des traces de variables, des commandes et les messages de l'interpréteur.

5.2.2. <u>Expérience sur deux écrans alphanumériques</u>

Nous utilisons pour cette expérience deux écrans de 28 lignes (SCORPION). Un des deux écrans est entièrement consacré au texte-programme, tandis que l'autre est partagé en trois fenêtres ; ces fenêtres sont respectivement affectées :

- à la structure du programme et à certains "paragraphes" du programme (listes de constantes, de types et de variables),
- aux entrées-sorties du programme interprété ainsi qu'aux traces,
- au dialogue.

Cette solution a plusieurs avantages. Tout d'abord, nous atteignons avec 28 lignes de texte affiché l'équivalent d'une demi-feuille de liste de programme. Ensuite, les informations relatives à la mise au point sont bien réparties et ne sont pas mélangées ; en effet, nous ne mélangeons que les traces et les éditions du programme, ce qui n'est pas trop gênant. Il est donc possible de visualiser des variables ou d'évaluer des expressions sans perdre la valeur des traces. Cette solution limite la taille de la partie affichée du programme, mais est très agréable pour travailler.

5.2.3. Expérience sur un écran bit-map

Nous avons réalisé une troisième expérience sur un buroviseur (<Naffah 79>). Ceci nous a apporté quelques renseignements intéressants sur le plan interface ; l'écran de type "bit-map" de 1024 X 768 points et disposé en hauteur nous a permis de découper l'écran en trois parties :

- La première occupe environ les deux tiers de l'écran et est découpée en deux fenêtres : une de quarante lignes contenant le texte source et l'autre contenant la structure du programme ou les traces de variables, suivant le cas.
- La partie médiane est réservée aux menus contenant les différentes fonctions de mise au point. Elle contient également une ligne permettant de rentrer des commandes ou bien le complément d'information nécessaire à la sélection d'une fonction du menu.
- Le bas de l'écran est la fenêtre dans laquelle le résultat des commandes ainsi que les entrées-sorties du programme en cours d'exécution sont affichés.

Cette expérimentation nous a permis de voir que la souris n'était utile que pour des commandes sans paramètre. En effet, le transit entre le clavier et la souris est très fastidieux ; de ce fait, l'utilisateur préfère en général entrer directement la commande au clavier. D'autre part, la lisibilité du programme est accrue avec l'augmentation considérable de la surface allouée au texte du programme ; toutefois quarante lignes restent quand même insuffisantes pour pouvoir mettre au point correctement son programme sans l'usage d'une liste.

CHAPITRE 4

Evaluation du système et conclusions

Pour cette thèse, nous avons été amené à réaliser deux prototypes. Le premier a été réalisé dans le cadre du projet ADELE. Ce dernier est réalisé sur l'ordinateur CII-HB 68 DPS 8 sous le système MULTICS. Le second, en cours, de réalisation est réalisé dans le cadre de l'atelier CONCERTO, projet du CNET (<André 82>). La suite de ce chapitre donne l'état du développement des deux systèmes et leurs caractéristiques spécifiques. Nous montrons ensuite les différentes contributions apportées et les limites de ces systèmes, puis nous énumérons les évolutions à envisager et les perspectives ouvertes par de tels outils.

1. ETAT DES DEVELOPPEMENTS

1.1. Le metteur au point de l'atelier ADELE

Nous ne reviendrons pas ici sur la description de cet environnement. Nous nous limitons à donner les caractéristiques techniques et plus particulièrement celles de l'interpréteur-metteur au point réalisé.

L'ensemble des outils de l'atelier sont écrits en PASCAL norme ISO, plus les extensions SOL en ce qui concerne la modularité. Ces différents composants sont implémentés sur le système MULTICS. Les postes de travail sont de simples terminaux alphanumériques type VT100 connectés au système hôte par des lignes asynchrones. Le prototype actuel n'intègre pas l'interface usager définie dans ce projet, car celle-ci est arrivée trop tard. Pour développer notre metteur au point, nous avons été amenés à nous définir un noyau d'exécution permettant de gérer l'enchainement des tâches relatives à l'édition syntaxique, la décompilation et la mise au point. Ce noyau est constitué de 400 lignes de code PASCAL. Les

procédures de gestion d'écran dépendantes du type de terminal constituent environ 300 lignes de programme par type de terminal. Toutefois le passage à l'interface usager ADELE devrait se faire sans trop de problèmes car toutes les procédures d'affichage et de saisie de commandes sont indépendantes des outils.

Le metteur au point réalisé offre les différentes fonctionnalités décrites dans le chapitre précédent sauf la manipulation d'environ. L'interpréteur permet l'exécution continue, visuelle ou en pas à pas de programme PASCAL norme ISO. L'interpréteur représente environ 4000 lignes de code PASCAL. Les différentes commandes du metteur au point occupent environ la même taille de code.

Ce prototype a montré la faisabilité du système et la capacité de développer des programmes sous ce metteur au point. Il est indéniable que les différentes informations sur le programme mises à la disposition de l'utilisateur lui permettent de développer rapidement une application. Toutefois, ce prototype n'est pas utilisable pour le développement de gros systèmes, car il ne prend pas en compte la modularité. Sans cette fonctionnalité, la programmation d'applications de taille moyenne est pratiquement irréalisable et notre outil ne peut être utilisé, par exemple, que pour l'apprentissage du langage par des étudiants.
De plus le dialogue entre l'éditeur et l'interpréteur n'est pas géré, ce qui impose une réexécution totale du programme après chaque modification.

Le défaut de ce système est également la syntaxe externe des commandes. En effet, le nom de chaque commande est constitué d'une seule lettre. Certaines de ces commandes sont facilement mémorisables par l'utilisateur car elles correspondent à des abbréviations classiques dans les systèmes (r pour run, g pour go, etc.), mais d'autres n'ont aucun rapport avec leur signification. Bien qu'une fonction d'aide soit présente et permette d'avoir la liste des commandes ou la syntaxe d'une commande particulière, il est à souhaiter une intégration avec l'interface ADELE.

Pour évaluer les performances de l'outil de mise au point, nous avons réalisé des mesures sur le temps d'exécution. Celles-ci

sont peu précises car il est difficile sous MULTICS d'évaluer le coût de prise en charge du système. Au vu de ces mesures, nous pouvons cependant dire qu'un programme interprété (réalisant des accès tableau, record, etc. et comportant l'exécution de fonctions récursives) met environ 50 fois plus de temps que le même programme compilé par le compilateur PASCAL actuel. Ces performances sont très acceptables compte tenu du fait que les programmes ne s'exécuteront jamais dans cet environnement, sauf durant la phase de mise au point. Bien que ce nombre soit impressionnant, il faut noter que le système s'exécutant en interactif offre des temps de réponse largement acceptables.

1.2. <u>Le metteur au point de l'environnement PASCAL CONCERTO</u>

Le projet CONCERTO s'est inspiré du projet ADELE démarré un an plus tôt. CONCERTO est un projet plus ambitieux prenant en compte différents poste de travail et la gestion de projet. L'édition de programme est paramétrée par la syntaxe du langage. Les postes de travail de cet atelier sont basés sur une représentation interne de type MENTOR (<Donzeau-Gouge 79-80>). La R.I est définie par un langage de spécification appelé METAL (<Mélèse 82>) et manipulée par un noyau de manipulation d'arbre (<Conchon 83>) constituant l'ensemble des procédures d'accès.

Cet environnement est réalisé sur des postes de travail individuels constitués de calculateurs SM90 auxquels sont connectés des écrans à points "bit-map" (1024 X 748) adressables par "souris". Le poste de travail PASCAL est écrit en LE_LISP (<Chailloux 82>). La programmation de ce système utilise également les facilités offertes par CEYX (<Hulot 83>). L'écran est géré par un gestionnaire de fenêtres appelé virtualiseur (<Cany 83>). Le noyau de synchronisation appelé dialogueur (<Rasser 84>) réalise le dialogue entre les différents outils. Il est écrit en ULYSSE (<Autret 83>), langage basé sur la théorie des acteurs et construit au dessus de LE_LISP.

Le changement de langage de programmation et de représentation interne nous a permis d'améliorer les différents modules. La réécriture a été facilitée par le choix du langage choisi par

CONCERTO. En effet, LISP est beaucoup mieux adapté que PASCAL pour ce genre d'application, car il se prête mieux aux traitements des listes et aux parcourts d'arbres. De plus, la notion d'annotations définie dans MENTOR rend plus souple la programmation du système. Les attributs spécifiques au metteur au point sont définis proprement et mémorisés dans ces annotations. Le code écrit correspond à environ 2500 lignes de LE_LISP pour l'interpréteur et autant pour le metteur au point, soit environ les deux tiers du code PASCAL. Ce prototype ne prend pas en compte la modularité et voit ainsi son intérêt limité. Les fonctionnalités décrites au chapitre III.4.1 sont toutes implémentées et le dialogue entre l'éditeur et l'interpréteur est géré. Celui-ci permet de désempiler les environs et relancer l'interprétation sur le premier bloc de la pile non affecté par les modifications. Ceci est d'un très grand intérêt, malgré le côté pénalisant d'une telle gestion d'environnement.

L'écriture de l'interpréteur-metteur au point pour ce nouveau système nous a permis d'expérimenter une autre interface. La sélection des commandes est faite par souris et les primitives de multi-fenêtrage de l'écran "bit-map" sont exploitées. Nous utilisons donc au maximum ces possibilités et avons alloué des fenêtres spécifiques aux différentes fonctions nécessitant des espaces d'affichages propres.

Ce système est actuellement très lent ; cette lenteur est due essentiellement au système ULYSSE. En effet, nous avons réalisé en premier lieu un système semblable à celui d'ADELE. Cet interpréteur s'exécutant directement sous LE_LISP offrait des temps d'exécution très honorables. L'introduction d'ULYSSE a considérablement augmenté ceux-ci. La cause de cette baisse de performance est du au fait suivant : pour donner à l'utilisateur la possibilité d'interrompre l'exécution de son programme, l'interpréteur doit régulièrement donner le contrôle au noyau ULYSSE afin que celui-ci traite une requête éventuelle de l'utilisateur. Ceci se traduit physiquement par une sauvegarde de tout l'environnement de l'interpréteur, une activation de l'acteur sollicité, puis une restauration de l'environnement avant de

relancer l'exécution.

Malgré ces performances, cette réalisation nous a permis d'explorer et d'appliquer les nouvelles techniques d'interface usager.

2. EVOLUTIONS SOUHAITEES

Pour rendre complets ces deux metteurs au point, il manque une fonctionnalité que nous n'avons pas retenue compte tenue de la structure des autres outils de l'atelier (introducteur et analyseur sémantique). Cette fonctionnalité devrait permettre une interprétation immédiate d'instructions frappées à la console par l'utilisateur. Ceci ne pose pas de problème majeur d'interprétation. En effet, les plus gros problèmes sont contextuels car, dans les deux systèmes, l'analyseur et le constructeur ne savent pas créer des arbres hors contexte. Une solution consisterait peut-être à rattacher le sous-arbre créé (correct syntaxiquement) à la première instruction "BEGIN" du programme. Après interprétation, ce sous-arbre serait détruit. Toutefois, si le programme est actif, il est peut-être souhaitable de rattacher ce sous-arbre au "BEGIN" du bloc de niveau supérieur (dernier bloc activé), mais ceci pose le problème de la visibilité des variables locales. Ceci permettrait en particulier d'évaluer des sous-expressions entrées directement au clavier.

Nous avons vu au chapitre III.3.7 que l'interpréteur que nous avons écrit possède une structure itérative. Ce choix a été fait à l'origine pour permettre une meilleure adaptabilité aux différentes architectures de systèmes. En effet, il peut être interrompu n'importe où et l'exécution peut se poursuivre ensuite si le programme est sémantiquement correct. Toutefois, ce mécanisme nous impose la gestion de nombreuses piles liées d'une part à l'exécution et d'autre part à l'évaluation d'expressions. Il est dommage de se priver de la récursivité offerte par les langages de programmation actuels pour gérer automatiquement ces

piles. Cependant, une programmation récursive impose certaines contraintes à l'environnement englobant (conservation de l'environ à la désactivation du bloc). Pour utiliser cette possibilité, deux types d'architecture sont possible.

L'interpréteur est activé en premier. Il fait alors appel à l'analyseur de commandes qui itère jusqu'à ce qu'une commande de réactivation de cet outil soit frappée. L'interprétation pourra alors commencer. Lorsqu'une condition d'arrêt sera détectée, l'interpréteur fera de nouveau appel à l'analyseur de commandes, et ainsi de suite. Dans ce scénario, l'interpréteur constitue l'outil principal du système, et celui-ci doit être systématiquement présent. C'est le fonctionnement type des interpréteurs LISP ou BASIC classiques.

La seconde solution consiste à lancer comme tâche principale le module chargé de l'interface usager (analyseur de commandes, gestion d'écran, etc.). Lorsque l'utilisateur donnera l'ordre d'interpréter le programme, la tâche interpréteur sera activée et ce jusqu'à ce qu'une condition d'arrêt soit remplie. Le processus sera alors suspendu et ne sera débloqué que sur une commande de réactivation venant de l'usager. Il est donc nécessaire d'avoir un système de gestion de processus. Cette solution est certainement la meilleure et la plus efficace car elle laisse l'interpréteur indépendant de l'atelier. Ces deux solutions simplifient grandement l'écriture des modules d'évaluation d'expressions, du noyau de l'interpréteur et du module chargé de la simulation des instructions. De plus, cette simplification doit certainement améliorer les temps d'exécutions du système. En effet, la prise en charge de la gestion des piles par le système (via la récursivité) doit être moins coûteuse que la gestion par programme car on peut supposer que les mécanismes liés à la récursivité sont des routines optimisées et généralement écrites en langage de bas niveau.

Enfin, pour être réellement utilisable, il est nécessaire de pouvoir exécuter des programmes constitués de modules interprétés et de modules compilés. Au fur et à mesure que l'utilisateur met au point des modules ou même des procédures, il les compile. Par

la suite, chaque référence à ces fonctions déclenchera le code machine correspondant et l'exécution sera beaucoup plus rapide. Ceci présente également un autre gros avantage : l'utilisation de modules écrits et compilés dont le code source associé ne correspond pas systématiquement au langage interprété. Cette solution très attrayante présente toutefois l'inconvénient d'être dépendante du système d'exploitation. En effet, le partage des données globales nécessite la connaissance et l'adaptation à l'éditeur de liens. De plus, l'édition de liens doit être dynamique pour permettre le chargement de modules à la demande.

Dans le cadre d'ADELE, il serait toutefois possible de réaliser un tel système à condition de conserver un langage unique. Dans ce cas, il serait possible de faire cohabiter le générateur de code et l'interpréteur. Les données partagées appartiendrait alors à une zone commune connue des deux outils, dans laquelle l'interpréteur rangerait ou consulterait les variables globales. Ceci nécessiterait alors un module de conversion permettant de transformer la valeur des variables codées de façon optimale dans la zone commune, en représentation propre à l'interpréteur.

3. PERSPECTIVES

Nos deux expériences montrent que des systèmes tels que ceux que nous avons réalisés sont promis à un bel avenir. Toutefois, pour une utilisation réelle en milieu industriel, il est nécessaire d'arriver à réduire les temps d'exécution de chaque composant. Ceci peut être fait en optimisant l'ensemble des modules et en utilisant du parallélisme entre les différents outils. Ceci passe évidemment par une architecture multiprocesseur du poste de travail et l'utilisation de processeurs spécialisés (en particulier pour l'affichage). Ceci devrait être atteint dans les années qui viennent, vu l'évolution des technologies. La baisse des coûts de ces machines (très gourmandes en ressources) devrait permettre une large diffusion de ces outils dans les prochaines années. Le gain de productivité ne sera obtenu que par l'intégration d'interpréteurs - metteurs au point qui ont

contribué à la popularité de certains langages de programmation. Enfin, ces systèmes permettront l'exécution de programmes constitués de R.I et de code binaire.

Il serait également souhaitable de rendre l'interpréteur indépendant du langage. Ceci est peu réalisable à l'heure actuelle car de tels outils doivent être paramétrés par une description de la sémantique du langage. Toutefois, dans un premier temps il est peut être envisageable de réaliser un interpréteur pour une classe de langages donnée. La réécriture d'un interpréteur pour un sous-ensemble du langage CHILL (<Chill 80>) devrait nous permettre de mettre en évidence les problèmes posés.

ANNEXES

ANNEXE 1

FONCTIONS DE MISE AU POINT

Fonction : Sortir du metteur au point avec réinitialisation des variables de travail (points d'arrêt, etc..).

Syntaxe : a

Fonction : Entrer en mode création de contexte.

Syntaxe : b paramètre
Valeur du paramètre : N° du contexte a créer.

Fonction : Décompile la structure du programme, d'une procédure, ou d'une fonction.

Syntaxe : S [paramètre]
Valeur du paramètre : Si le paramètre est absent, on décompile la structure du programme. Sinon on décompile la structure de la procédure ou fonction désigné par le N° de noeud correspondant.

Fonction : Décompiler le sous arbre désigné.

Syntaxe : Z [paramètre]
Valeur du paramètre : N° du noeud de départ ; par défaut cette valeur est 1.

Fonction : Sortir du mode création de contexte.

Syntaxe : q

Fonction : Ajouter une commande dans un contexte.

Syntaxe : o paramètre1 paramètre2
Valeur des paramètres :
paramètre1 : N° de contexte.
paramètre2 : N° de ligne après laquelle sera inséré la commande.

Fonction : Supprimer une commande d'un contexte.

Syntaxe : k paramètre1 paramètre2
Valeur des paramètres :
paramètre1 : N° de contexte.
paramètre2 : N° de ligne à supprimer dans le contexte désigné.

Fonction : Afficher un contexte.

Syntaxe : d [paramètre]
Valeur du paramètre : Si le paramètre est omis, on imprime les commandes contenues dans le contexte actif sinon le paramètre désigne le contexte à visualiser.

Fonction : Utiliser un contexte.

Syntaxe : u paramètre
Valeur du paramètre :
N° du contexte activé chaque fois que la main sera rendue à l'utilisateur.

Fonction : Fin d'utilisation d'un contexte.

Syntaxe : f

Fonction : Exécuter un contexte.

Syntaxe : x paramètre
Valeur du paramètre
N° du contexte qu'on veut exécuter une fois.

Fonction : Visualiser la pile d'exécution.

Syntaxe : y

Fonction : Changer la valeur d'une variable du programme.

Syntaxe : c paramètre1 = paramètre2
Valeur des paramètres :
paramètre1 : nom de l'identificateur
paramètre2 : valeur à donner au paramètre1.

Fonction : Evaluer la partie expression d'une instruction.

Syntaxe : e [paramètre]
Valeur du paramètre :
Numéro du noeud instruction sur lequel porte l'évaluation : si ce numéro est omis on évaluera le noeud courant.

Fonction : Lancer l'interprétation à partir du noeud courant.

Syntaxe : g

Fonction : Donne la syntaxe d'une commande.

Syntaxe : h [paramètre]
Valeur du paramètre :
Si le paramètre est précisé, on éditera les fonctions, syntaxes, etc. de la commande désignée, sinon on donnera la liste des commandes disponibles.

Fonction : Active l'interpréteur au début du programme.

Syntaxe : i

Fonction : Liste une fenêtre.

Syntaxe : l [noeud de début]
Valeur du paramètre :
numéro du noeud de début. Si ce numéro est omis on listera à partir du noeud courant. Si le programme n'est pas actif on listera à partir du noeud program. Si le paramètre vaut -1 on listera la fenêtre précédente.

Fonction : Mémoriser une fenêtre affichée.

Syntaxe : m ['RETURN'] puis numéro de case
Valeur du paramètre : numéro entre 1 et 10
Remarque : Pour ne pas afficher le tableau il suffit de ne pas taper de RETURN avant le numéro de case.

Fonction : Désactive le pas à pas.

Syntaxe : n

Fonction : Passe en mode pas à pas et exécute un pas de programme.

Syntaxe : p

Fonction : Enlever un point d'arrêt.

Syntaxe : r [paramètre1] .. [paramètreN]
Valeur du paramètre :
Si la liste de points d'arrêt est présente
on supprime les points d'arrêt désignés, sinon
on donne la liste des points d'arrêt.

Fonction : Mettre un point d'arrêt.

Syntaxe : s paramètre1 [paramètre2]
Valeur des paramètres :
paramètre1 : adresse du point d'arrêt
paramètre2 : si ce paramètre est présent, on affectera
la valeur donnée au point d'arrêt. Dans ce cas
on s'arrêtera sur l'instruction après N passages.
Si le paramètre2 est absent la valeur par défaut est 1.

Fonction : Afficher une fenêtre mémorisée.

Syntaxe : t ['RETURN'] numéro de case
Valeur du paramètre : nombre de 1 à 10
Remarque : Pour ne pas afficher le tableau il suffit de ne pas
taper le RETURN avant le numéro de case.

Fonction : Visualiser une variable.

Syntaxe : v paramètre

Valeur du paramètre : Nom d'un identificateur du programme. Si l'identificateur est précédé de "<" on visualisera une variable globale. Si l'identificateur est un tableau il doit être suivit de : [indice, [indice]] ou indice peut être : un entier, *, un intervalle ou un identificateur.

Fonction : Afficher une instruction.

Syntaxe : w [paramètre]

Valeur du paramètre : N° du noeud. Si le paramètre est absent, on affichera le noeud courant.

Fonction : Enlève tous les points d'arrêt du programme.

Syntaxe : z

Fonction : Exécuter une commande Multics.

Syntaxe : . [commande Multics]

Remarque : Si "." n'est pas suivit d'une commande, on retourne au niveau ADELE, sans réinitialiser le MAP.

ANNEXE 2

SYNTAXE DE LA RI ADELE

Notation : Les mots en minuscules correspondent à des noms de sommet, les mots en majuscules à des noms de classe.

Définition d'un sommet :

```
xyz -->attribut1,...,attributn;
    ==>fils1,...,filsp;
```

Définition d'une classe :

```
XYZ ::= sommet1 ! sommet2 ! ...
```

Un fils optionnel est entouré de crochets [ ].
La racine de l'arbre est le sommet racine.

1. CLASSES

```
BORNE           ::= declidf constentier;
CONST           ::= CTE opconst constchaine constnil;
CTE             ::= constentier constreel utilidf;
DEBUTPROG       ::= prog interface corps;
DECLIDF         ::= declidf  predefidf ;
DECLOUDIR       ::= DECLIDF directive;
DECLPROCFONCT   ::= declproc declfonct directive;
DEFPARAM        ::= lparam;
DESCRIPTEUR     ::= TYPORDINAL TYPESTRUCTURE utilidf pointeur;
ELEMENS         ::= EXP elemens;
ENONCE          ::= INSTR etiqinstr;
EXP             ::= CONST REFERENCE opbinaire opunaire appelfonct
                    constrens ensvide;
```

```
IDF             ::= declidf  utilidf predefidf indefidf;
INSTR           ::= instraffect instrarret instrassert instrbegin
                    instrcase instrfor instrgoto instrif instrrepeat
                    instrwhile instrwith appelproc instrvide;
PARAMETRE       ::= parvaleur parvariable parproc parfonct
                    partableau;
PAREFFECTIF     ::= EXP writepar;
PARGEN          ::= constgen privategen rangegen;
PARGENEFFECTIF::= utilidf CONST;
REFERENCE       ::= utilidf accestab acceschamp indirect tampon
                    withchamp;
REGION          ::= DEBUTPROG racine declproc declfonct lpredefini
                    lindef article;
TYPECONSTGEN    ::= utilidf interv chainegen;
TYPELEMENT      ::= utilidf tabpacke tabnonpacke;
TYPENONPACKE    ::= tableau article ensemble fichier;
TYPORDINAL      ::= utilidf enum interv;
TYPESTRUCTURE ::= typepacke TYPENONPACKE;
TYPETAB         ::= tabpacke tabnonpacke;
```

2. SOMMETS

```
acceschamp       --> erreur,base,contrainte
                 ==> REFERENCE,utilidf;

accestab         --> erreur,base,contrainte
                 ==> REFERENCE,EXP;

appelfonct       --> erreur,base;
                 ==> utilidf,[lpareffectif];

appelproc        --> erreur
                 ==> utilidf,[lpareffectif];

article          --> erreur,typelem,base,contrainte,max
                 ==> [lchamp],[partievariante];
```

```
bloc            --> erreur
                ==> [letiq],[lcte],[ltype],[lvar],[lprocfonct],
                    instrbegin;

blocint         --> erreur
                ==> [lcte],[ltype],[lprive],[llimite],[lvar],
                    [lronly],[lprocfonct];

chainegen       --> erreur,max
                ==> BORNE;

chaineint       --> erreur, chaine,versinterface
                ==> ;

champ           --> erreur,typelem
                ==> ldeclidf,DESCRIPTEUR;

comment         --> erreur,chaine,souscomment;
                ==> ALL;

constrens       --> erreur,base
                ==> seq of ELEMENS;

constchaine     --> erreur,chaine,base,valentier
                ==> ;

constentier     --> erreur,chaine,valentier,base
                ==> ;

constgen        --> erreur
                ==> ldeclidf,TYPCONSTGEN;

constnil        --> erreur,base
                ==> ;
```

```
constreel       --> erreur,chaine,valreel,base
                ==> ;

corps           --> erreur,chaine
                ==> idinterface,[lparprog],[limport],[lnew],bloc;

create          --> erreur
                ==> declidf,chaineint,[lpargeneff];

declconst       --> erreur
                ==> declidf,CONST;

declfonct       --> erreur,sousproc
                ==> declidf,[lparam],utilidf,bloc;

declidf         --> erreur,chaine,sousidf,base,contrainte,
                    region,numregion,numdecl;
                ==> ;

declproc        --> erreur,sousproc
                ==> declidf,[lparam],bloc;

decltype        --> erreur
                ==> declidf,DESCRIPTEUR;

declvar         --> erreur
                ==> ldeclidf,DESCRIPTEUR;

directive       --> erreur,objetdirect,sousdirective
                ==> ;

elemcase        --> erreur
                ==> lconst,ENONCE;

elemens         --> erreur,base
                ==> EXP,EXP;
```

```
ensemble        --> erreur,base,contrainte,typelem,max
                ==> TYPORDINAL;

ensvide         --> erreur,base
                ==> ;

enum            --> erreur,min,max,base,contrainte
                ==> seq of declidf;

etiqinstr       --> erreur
                ==> declidf,INSTR;

fichier         --> erreur,base,contrainte,typelem
                ==> DESCRIPTEUR;

idinterface     --> erreur,chaine,versinterface
                ==> ;

indefidf        --> erreur,chaine,region
                ==> ;

indice          --> erreur,base,contrainte
                ==> declidf,declidf,utilidf;

indirect        --> erreur,base,contrainte
                ==> REFERENCE;

instraffect     --> erreur,base,contrainte
                ==> REFERENCE,EXP;

instrarret      --> erreur
                ==> ENONCE,[EXP];

instrassert     --> erreur
                ==> lenonce,EXP;
```

```
instrbegin      --> erreur
                ==> seq of ENONCE;

instrcase       --> erreur
                ==> EXP,lelemcase;

instrfor        --> erreur,base,contrainte,sousfor
                ==> utilidf,EXP,EXP,ENONCE;

instrgoto       --> erreur
                ==> utilidf;

instrif         --> erreur
                ==> EXP,ENONCE,[ENONCE];

instrrepeat     --> erreur
                ==> lenonce,EXP;

instrvide       --> erreur
                ==> ;

instrwhile      --> erreur
                ==> EXP,ENONCE;

instrwith       --> erreur
                ==> lref,ENONCE;

interface       --> erreur
                ==> idinterface,[lpargen],blocint;

interv          --> erreur,base,contrainte,min,max,typelem
                ==> CONST,CONST;

lchamp          --> erreur
                ==> seq of champ;
```

```
lconst          --> erreur
                ==> seq of CONST;

lcte            --> erreur
                ==> seq of declconst;

ldeclidf        --> erreur
                ==> seq of DECLOUDIR;

lelemcase       --> erreur
                ==> seq of elemcase;

lenonce         --> erreur
                ==> seq of ENONCE

letiq           --> erreur
                ==> seq of utilidf;

limport         --> erreur
                ==> seq of create;

lindef          --> erreur
                ==> seq of indefidf;

linterface      --> erreur
                ==> seq of DEBUTPROG;

llimite         --> erreur
                ==> seq of IDF;

lnew            --> erreur
                ==> seq of create;

lparam          --> erreur
                ==> seq of PARAMETRE
```

```
lpareffectif    --> erreur
                ==> seq of PAREFFECTIF;

lpargen         --> erreur
                ==> seq of PARGEN;

lpargeneff      --> erreur
                ==> seq of PARGENEFFCTIF;

lparprog        --> erreur
                ==> seq of IDF;

lpredefini      --> erreur
                ==> seq of predefidf;

lprive          -->erreur
                ==> seq of IDF;

lprocfonct      --> erreur
                ==> seq of DECLPROCFONCT;

lref            --> erreur
                ==> seq of REFERENCE;

lronly          --> erreur
                ==> seq of utilidf;

ltype           --> erreur
                ==> seq of decltype;

lvar            --> erreur
                ==> seq of declvar;

lvariante       --> erreur
                ==> seq of variante;
```

```
opbinaire        --> erreur,sousopbinaire,base
                 ==> EXP,EXP;

opconst          --> erreur,sousopconst,base
                 ==> CTE;

opunaire         --> erreur,sousopunaire,base
                 ==> EXP;

parfonct         --> erreur
                 ==> declidf,[lparam],utilidf;

parproc          --> erreur
                 ==> declidf,[lparam];

partableau       --> erreur
                 ==> ldeclidf,TYPETAB;

partievariante   --> erreur
                 ==> [declidf],utilidf,lvariante;

parvaleur        --> erreur
                 ==> ldeclidf,utilidf;

parvariable      --> erreur
                 ==> ldeclidf,utilidf;

pointeur         --> erreur,base,contrainte,typelem
                 ==> utilidf;

predefidf        --> erreur,chaine,region,souspredefidf,
                     base,contrainte,sousidf,numdecl
                 ==> ;

privategen       --> erreur
                 ==> seq of IDF;
```

```
prog            --> erreur,chaine
                ==> lparprog, bloc;

racine          --> erreur
                ==> linterface,lindef,DEBUTPROG;

rangegen        --> erreur,min,max
                ==> declidf,BORNE,BORNE;

tableau         --> erreur,base,contrainte,typelem,max
                ==> TYPORDINAL,DESCRIPTEUR;

tabnonpacke     --> erreur,base,contrainte,typelem,max
                ==> indice,TYPELEMENT;

tabpacke        --> erreur,base,contrainte,typelem,max
                ==> indice,utilidf;

tampon          --> erreur,base,contrainte
                ==> REFERENCE;

typepacke       --> erreur,string,typelem,max,base,contrainte
                ==> TYPENONPACKE;

utilidf         --> erreur,declar
                ==> ;

variante        --> erreur
                ==> lconst,[lchamp],[partievariante];

vide            --> erreur
                ==> ;

withchamp       --> erreur,refwith,base,contrainte
                ==> utilidf;
```

writepar --> erreur,base
==> EXP,[EXP],[EXP];

3. ATTRIBUTS

base: liaison vers un sous-arbre constructeur de type ou un identificateur de type prédéfini.

chaine: chaîne de caractères
EX: "ab""cd""" = ab"cd".

contrainte : pointe vers un sommet interv, n'a de sens que pour les types ordinaux et tableaux ; (si type tableau alors c'est la contrainte sur l'indice).

declar : pointe vers la déclaration de l'idf, de classe DECLIDF.

erreur : numéro d'erreur
0: pas d'erreur
1: non analyse
>1: numero d'erreur.

min : valeur minimale d'un intervalle.

max: valeur maximale d'un intervalle ou taille d'un type à l'interprétation.

numregion : entier (numéro de région à l'interprétation).

numdecl : entier (numéro de déclaration à l'interprétation).

objetdirect : pointe sur l'objet de la directive.

refwith : pointe vers un sommet de classe REFERENCE associe à un instrwith.

région : pointe vers la région de déclaration de l'idf, de classe REGION.

souscomment : entier.

sousdirective : entier.

sousfor : énumération (forto,fordownto).

sousidf : énumération
(etiq,const,type,var,proc,fonct,champ,parval,
parvar,parproc,parfonct,partableau,input,output,
erreur,borne,interface,genconst,gentype,typeprive,
typelimite,ronly).

sousopbinaire : énumération
(+,-,*,/,div,mod,and,or,=,<>,< ,<=,> ,>=,in).

sousopconst : énumération (+,-).

sousopunaire : énumération (+,-,not,conversion).

souspredefidf : énumération
(abs,arctan,boolean,char,chr,cos,dispose,eof,
eoln,exp,false,get,integer,ln,maxint,new,
odd,ord,pack,page,pred,put,read,readln,real,
reset,rewrite,round,sin,sqr,sqrt,succ,text,
true,trunc,unpack,write,writeln,argc,argv,
fappend,fconnect,fget,fllength,flush,fpos,
fput,fsize,fstatus,freopen,fupdate,maxreal,
minreal,setmax,sread,stop,swrite).

sousproc : entier (type de la déclaration).

string: entier (0 si non de type chaîne, taille de la chaîne sinon).

typelem : entier (0: éléments normaux,
1: il y a un élément de type fichier,
2: il y a un élément de type limité privé).

valentier : entier (valeur d'une constante entière).

valreel : réel (valeur d'une constante réelle).

versinterface : pointe sur un sommet interface.

ANNEXE 3

INTERFACE RI MEMOIRE : PROCEDURES

```
procedure init (var f : text) ;
   (* initialise les sommets prédéfinis à partir du fichier f *)

function creer (nom : sommet) : liaison ;
   (* crée un sommet de type nom ;   *)
   (* tous les attributs valent null *)

function creervide (nom : sommet) : liaison ;
   (* crée un sommet de type nom ; tous ses fils obligatoires *)
   (* sont créés (sommets vide)                               *)

procedure detruire (adr : liaison) ;
   (* détruit le sommet adr; ses fils ne sont pas détruits *)

procedure detruirarbre (adr : liaison) ;
   (* détruit le sous-arbre de racine arbre *)

function null : liaison ;
   (* retourne la valeur "nil" *)

function nullouvide (adr : liaison) : boolean ;
   (* retourne vrai si adr a la valeur "nil" *)
   (* ou si typesommet (adr) = vide          *)

procedure lier (adpere, adfils : liaison ; att : attribut) ;
   (* le sommet adfils devient le att-ieme fils de adpere *)

function fils (adpere : liaison ; att : attribut) : liaison ;
   (* retourne l'att-ieme fils de adpere *)
```

```
function pere (adfils : liaison) : liaison ;
   (* retourne le père de adfils *)

function frere (adfils : liaison ; att : attribut) : liaison ;
   (* frere (a,n) = fils (pere (a), n) *)

function typesommet (adr : liaison) : sommet ;
   (* retourne le type du sommet adr *)

function typattribut (adr : liaison) : attribut ;
   (* typattribut (a) = n tel que fils (pere (a), n) = a *)

(************************)
(* TRAITEMENT DE LISTES *)
(************************)

function premier (adliste : liaison) : liaison ;
   (* retourne le premier élément de la liste adliste *)

function suivant (adr : liaison) : liaison ;
   (* retourne l'élément suivant de adr *)
   (* dans la liste pere (adr)         *)

function precedent (adr : liaison) : liaison ;
   (* retourne l'élément précédent de *)
   (* adr dans la liste pere (adr)    *)

function dernier (adliste : liaison) : liaison ;
   (* retourne le dernier élément de la liste adliste *)

procedure inspremier (adliste, adr : liaison) ;
   (* insère l'élément adr en tête de la liste adliste *)

procedure insdernier (adliste, adr : liaison) ;
   (* insère l'élément adr en queue de la liste adliste *)
```

```
procedure inssuivant (adr1, adr2 : liaison) ;
   (* insère adr2 après adr1 dans la liste pere (adr1) *)

procedure insprecedent (adr1, adr2 : liaison) ;
   (* insère adr2 avant adr1 dans la liste pere (adr1) *)

procedure oter (adr : liaison) ;
   (* enlève l'élément adr de la liste pere(adr) *)

function arite (adliste : liaison) : integer ;
   (* retourne le nombre d'éléments de la liste adliste *)

(********************************)
(* MANIPULATION DES ATTRIBUTS *)
(********************************)

function cdeclar (adr : liaison) : liaison ;
   (* retourne le sommet pointe par l'attribut declar *)

function cbase (adr : liaison) : liaison ;

function ccontrainte (adr : liaison) : liaison ;

function centete (adr : liaison) : liaison ;

function crefwith (adr : liaison) : liaison ;

function cpremref (adr : liaison) : liaison ;

function csuivdecl (adr : liaison) : liaison ;

function cregion (adr : liaison) : liaison ;

function csuivref (adr : liaison) : liaison ;

function cprecref (adr : liaison) : liaison ;
```

```
function cversinterface (adr : liaison) : liaison ;

function cobjetdirect (adr : liaison) : liaison ;


(* attributs de type entier ou enumeration *)
(*----------------------------------------*)

function cnumboite (adr : liaison) : integer ;

function cerreur (adr : liaison) : integer ;

function csouscomment (adr : liaison) : integer ;

function csousdirective (adr : liaison) : integer ;

function csousproc (adr : liaison) : liaison ;

function cstring (adr : liaison) : integer ;

function ctypechamps (adr : liaison) : integer ;

function cmin (adr : liaison) : liaison ;

function cmax (adr : liaison) : integer ;

function cvalentier (adr : liaison) : integer ;

function cvalreel (adr : liaison) : real ;

function cnumregion (adr : liaison) : integer ;
   (* numero de la région de declaration, pour l'interpréteur *)

function cnumdecl (adr : liaison) : integer ;
   (* numéro de la déclaration dans la région courante *)
   (* pour l'interpréteur                              *)
```

```
function csousfor (adr : liaison) : sousfor ;

function csousidf (adr : liaison) : sousidf ;

function csousopbinaire (adr : liaison) : sousopbinaire ;

function csousopconst (adr : liaison) : sousopconst ;

function csousopunaire (adr : liaison) : sousopunaire ;

function csouspredefidf (adr : liaison) : souspredefidf ;

procedure cchaine (adr : liaison ; var valchaine : chaine) ;

function lgchaine (adr : liaison) : integer ;

(***********************)
(* procedures diverses *)
(***********************)

function adpredef (valpredef : souspredefidf) : liaison ;
   (* retourne l'adresse du sommet prédéfini specifie *)

function adlistepredef : liaison ;
   (* retourne l'adresse du sommet lpredef *)

procedure remplacer(adr1,adr2:liaison);
   (* remplace adr1 par adr2 *)
```

BIBLIOGRAPHIE

BIBLIOGRAPHIE

<ACM 83>
ACM, "ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Higth Level Debugging"
Sigplan Notices 18
Août 1983

<Aho 79>
Aho A.V. Ullman J.D.
"Principles of compiler design"
Addison-Wesley Publishing Company
New Jersey. Avril 1979

<Andre 82>
Andre E.
"Présentation du projet pilote CONCERTO"
Journées Bigre
Grenoble, Janvier 1982, pp. 231-242

<Autret 83>
Autret Y.
"ULYSSE : Manuel de référence"
CNET Lannion, Décembre 1983

<Briat 81>
Briat J. et al
"Adèle : Un atelier de développement de logiciel"
AFCET Informatique
Paris, Novembre 1981, pp. 189-199

<Cany 83>
Cany G.
"Spécification du virtualiseur Concerto"
Mélodie, Mars 1983

<Cassagne 82>

Cassagne B., Hochain J.C., Santana M.
"Génération de code multicible"
Journées d'études CONCERTO
Perros Guirec, Décembre 1982

<Chailloux 83>

Chailloux J.
Le_Lisp : Manuel de référence
INRIA, Novembre 1983

<Cheval 82>

Cheval J.L, Estublier J., Ghoul S., Krakowiak S.
"Modularité et composition des programmes dans l'atelier de logiciel Adèle"
Colloque Génie Logiciel (AFCET)
Paris, Juin 1982, pp. 183-197

<Chill 80>

CCITT
"CHILL : Reference Manual"
Genève, Mai 1980

<Conchon 83>

Conchon A., Lang B.
"Spécification d'un noyau de manipulation d'arbres pour un environnement de programmation"
Journées BIGRE 83
Le Cap d'Agde, Octobre 1983, pp. 617-628

<Coutaz 83>

Coutaz J., Herrmann M.
"Adèle et le médiateur-compositeur, ou comment rendre une application interactive indépendante de l'interface usager".
Journées BIGRE 83
Le Cap d'Agde, Octobre 1983, pp. 1-17

<Donzeau-Gouge 79>
Donzeau-Gouge V., Huet G., Kahn G.,Lang B.
"Introduction au système Mentor et à ses applications"
Journées francophones sur la certification du logiciel
Genève, Janvier 1979

<Donzeau-Gouge 80>
Donzeau-Gouge V., Huet G., Kahn G.,Lang B.
"Programming environments based on stuctured editors : the MENTOR experience"
Rapport INRIA 26, Juillet 1980

<Estublier 83>
Estublier J., Krakowiak S., Mossière J., Rouzaud Y.
"Design principles of the Adele programming environment"
7th International Computing Symposium
Nuremberg, Mars 1983

<Estublier 84>
Estublier J., Ghoul S.
"Un système automatique de gestion de gros logiciels : la base de programmes Adèle"
Deuxième colloque de Génie Logiciel (AFCET)
Nice, Juin 1984

<Finger 82>
Finger U. et al
"Description générale de la SM90"
Note technique NT/PAA/OGE/SML/703
CNET Lannion, Mai 1982

<Fritzson 82>
Fritzson P.
"Fine-grained Incremental Compilation for PASCAL-like Languages"
LITH-MAT-R-82-15
Software Systems Research Center
Linkoping University
Linkoping (Sweden), Juillet 1982

<Fritzson 83>
Fritzson P.
"A Systematic Approach to Advanced Debugging through Incremental Compilation"
LITH-MAT-R-83-12
Software Systems Research Center
Linkoping University
Linkoping (Sweden), Avril 1983

<Ghoul 83>
Ghoul S.
"Base de données et gestion de configurations dans un atelier de génie logiciel"
Thèse Docteur-Ingénieur
Grenoble, Décembre 1983

<Habermann 82>
Habermann N., Ellison R., Medina-Mora R., Feiler P., Notkin D., Kaiser G., Gerlan D., Popovitch S.
"The second Compendium of Gandalf Documentation"
TR CS-82
Carnegie-Mellon University, Mai 1982

<Herrmann 82>
Herrmann M., Raymond J.
"Le poste de travail Adèle"
In "Adèle : Un atelier de développement de logiciel"
Laboratoire IMAG R.R. 299
Grenoble, 1982

<Herrmann 84>
Herrmann M.
"Interface usager-application dans un atelier de génie logiciel"
Thèse 3ème Cycle
Grenoble, Juillet 1984

<Hulot 83>
Hulot J.M.
"Ceyx, a multiformalism environment"
Rapport INRIA 210, Mai 1983

<ICL 82>
"ICL Perq : System Software Reference"
RP 10101
Aout 1982, pp 2.57-2.59

<Jensen 77>
Jensen K., Wirth N.
"Pascal User Manual and Report"
Springer Verlag, 1977

<Maclisp 78>
"Maclisp Reference Manual"
Juillet 1978

<Mélèse 82>
Mélèse B.
"Métal : un langage de spécifications pour le langage Mentor"
Technique et Science Informatiques, Vol. 1(4)
1982, pp. 275-286

<Mikelsons 80>
Mikelsons M., Wegman M. N
"PDE1L : The PL1L Program Development Environment
Principles of Operation"
IBM Research Report RC 8513 (37030)
Yorktown Heights, New York, Septembre 1980

<Mikelsons 80a>
Mikelsons M.
"Lispedit Command Descriptions"
IBM Research Report RC 8275 (35994)
Yorktown Heights, New York, Mai 1980

<Mossière 82>

Mossière J., Raymond J., Rouzaud Y.
"Représentation interne et manipulation de programmes dans l'atelier de logiciel Adèle"
Premier Colloque Génie Logiciel (AFCET)
Paris, Juin 1982, pp. 137-150

<Myers 80>

Myers B. A.
"Displaying Data Structures for Interactive Debbuging"
Palo Alto Research Center : Xerox PARC, Juin 1980

<Naffah 79>

Naffah N.
"Exemple d'un poste de travail burotique à interface universelle"
T.R. MEV.2.503, Projet pilote KAYAK
INRIA, Juin 1979

<Probe 83>

"Multics : Commands and Active Functions"
CII-HB Louveciennes, Avril 1983, pp. 3.612-3.636

<Rasser 84>

Rasser A.M.
"Spécifications du dialogueur Concerto"
CNET Paris A, Mai 1985

<Rouzaud 84>

Rouzaud Y.
"Représentation et manipulation de programmes dans un atelier de génie logiciel"
Thèse de Docteur-Ingénieur
Grenoble, Juin 1984

<Sackman 68>

Sackman H., Erikson W.J., Grant E.E.
"Exploratory Experimental Studies Comparing On-line and Off-line Programming Performance"
Communications of the ACM, vol.11, 1 (Janvier 1968), pp. 3-11

<Santana 83>

Santana M.
"Un système de production automatique de générateurs de code"
Thèse de 3ième cycle
Grenoble, Décembre 1983

<Teitelbaum 80>

Teitelbaum T., Reps T.
"The Cornell Program Synthesiser : a syntax-directed programming environment"
Cornell University, Mai 1980

<Teitelman 78>

Teitelman W.
"Interlisp Reference Manual"
Palo Alto : Xerox Parc, 1978

<Thomson 82>

"Micromega-32. Guide utilisateur"
ThomsonCSF. Département Informatique de Bureau
Paris, 1982

<Unix 79>

"Unix Programmer's Manual : ADB"
Bell Telephone Laboratories
New Jersey, Février 1979

<Wertz 84>

Wertz H.
"Etude, Réalisation et Evaluation d'un environnement de programmation utilisant des Représentations multiples pour le Développement continu de Logiciels très évolués"
Thèse d'état, Université Paris 8 - Vincennes, Avril 1984

DERNIERE PAGE D'UNE THESE

3È CYCLE, DOCTEUR INGÉNIEUR OU UNIVERSITÉ

Vu les dispositions de l'arrêté du 16 avril 1974,

Vu les rapports de M. MOSSIERE J. ....

M. ....

M. LENNE Christian .... est autorisé

à présenter une thèse en vue de l'obtention du grade de DOCTEUR de 3ème Cycle.

....

Grenoble, le 13 MAI 1985

Le Président de l'Université Scientifique et Médicale

M. TANCHE

RESUME

Cette thèse présente différents types de metteurs au point, puis elle essaye d'apporter une solution à ce problème en présentant le metteur au point PASCAL réalisé au sein de l'environnement de programmation ADELE. Pour cela, elle décrit l'interpréteur réalisé dans le cadre de ce projet ; cet interpréteur travaille sur une représentation interne de l'arbre abstrait des programmes. Nous montrons ensuite comment est exploité cet outil pour réaliser les différentes fonctions de mise au point offertes à l'utilisateur.

MOTS CLES

Génie logiciel, mise au point, interpréteur, environnement de programmation.