Représentation et manipulation de programmes dans un atelier de génie logiciel

Yann Rouzaud

THESE

présentée à

l' Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR INGENIEUR
INFORMATIQUE

par

Yann ROUZAUD

REPRESENTATION ET MANIPULATION DE PROGRAMMES DANS UN ATELIER DE GENIE LOGICIEL

Thèse soutenue le 15 juin 1984 devant la Commission d'Examen :

Monsieur	J. MOSSIERE	: Président
Messieurs	S. KRAKOWIAK	Examinateurs
	E. ANDRE	
	G. KAHN	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

à Catherine

Lors de l'écriture d'une thèse, il arrive un moment où, le travail presque terminé, la décontraction est suffisamment importante pour que la notion de remerciements prenne son véritable sens, sincère et chaleureux.

C'est pourquoi je tiens à remercier

Jacques Mossière, Directeur du Laboratoire de Génie Informatique, de m'avoir prodigué conseils et encouragements aux moments où j'en avais le plus besoin. C'est avec plaisir que je le remercie d'avoir accepté de présider le jury de cette thèse.

Sacha Krakowiak, Professeur à l'Université de Grenoble, d'avoir su diriger la conduite de ma thèse avec le soin qu'il fallait, et d'avoir ainsi guidé mes premiers pas dans la recherche.

Edouard André, Directeur du projet Concerto, d'avoir partiellement assuré le financement du projet Adèle, ce qui m'a permis d'expérimenter mes propositions. Je tiens de plus à le remercier d'avoir écrit avec tant de bienveillance et de célérité le rapport de présentation de ma thèse.

Gilles Kahn, Ingénieur INRIA, dont les travaux sur le système Mentor ont eu une grande influence sur la conduite de ma recherche.

Tous les membres de l'équipe Adèle, qui ont toujours su apporter des éléments critiques lors des multiples réunions de travail, et qui ont également su discuter de la montagne.

Le sous-groupe "Représentation Interne" du projet Concerto, et particulièrement Bernard Lang et Alain Conchon, qui m'ont permis de mieux comprendre les problèmes à résoudre.

Fabienne Carrier qui, lors de son projet de DEA, m'a sans cesse invité à remettre en question mes vues sur l'analyse incrémentale.

Geneviève Boulesteix, d'avoir assuré avec efficacité la saisie de mon manuscrit, sans jamais pester contre mon écriture.

Daniel Iglésias et son équipe de reprographie, d'avoir apporté leur soin légendaire à la présentation matérielle de cette thèse.

Les officiers de l'Armée de l'Air qui m'ont accueilli, et en particulier le Capitaine Jourdan, qui m'a accordé les jours de permission nécessaires pour rédiger mon manuscrit tout au long de mon année de service militaire.

REPRESENTATION ET MANIPULATION DE PROGRAMMES

DANS UN ATELIER DE GENIE LOGICIEL

INTRODUCTION. 1

CHAPITRE 1 : ATELIERS DE GENIE LOGICIEL. 3
- 1.1. INTRODUCTION. 3
- 1.2. AIDE A L'ECRITURE D'UN COMPOSANT LOGICIEL. 5
  - 1.2.1. Les limites du schéma classique d'écriture. 5
  - 1.2.2. Réduction du cycle d'écriture. 7
- 1.3. PRESENTATION DU PROJET ADELE. 9
  - 1.3.1. Objectifs. 9
  - 1.3.2. Extensions au langage Pascal. 11
  - 1.3.3. Archivage des modules. 14
  - 1.3.4. Interpréteur. 16
  - 1.3.5. Interface usager-application. 18
  - 1.3.6. Compositeur. 19
  - 1.3.7. Médiateur. 21

CHAPITRE 2 : PRINCIPES DE L'EDITION SYNTAXIQUE. 23
- 2.1. PRESENTATION GENERALE. 23
- 2.2. MENTOR. 26
- 2.3. CPS. 31
- 2.4. DISCUSSION. 36
- 2.5. L'EDITEUR SYNTAXIQUE DE L'ATELIER ADELE. 40

CHAPITRE 3 : REPRESENTATIONS INTERNES DE PROGRAMMES ET ARBRES ABSTRAITS. 47
- 3.1. CRITERES DE CHOIX D'UNE RI. 48
- 3.2. QUELQUES CHOIX POSSIBLES. 49
- 3.3. RI FONDEES SUR LES ARBRES ABSTRAITS. 53
  - 3.3.1. Introduction. 53
  - 3.3.2. Algèbre des ramifications et bigrammaires régulières. 54
  - 3.3.3. 1ère définition de grammaire abstraite. 57
  - 3.3.4. 2ème définition de grammaire abstraite. 61
  - 3.3.5. Arbres abstraits et annotations. 63

3.4. PROBLEMES LIES A L'ELABORATION D'UNE GRAMMAIRE ABSTRAITE. 65
3.4.1. Arbre abstrait et informations contextuelles. 66
3.4.2. Ambiguïtés syntaxiques. 67
3.4.3. Exemples divers. 69
3.4.4. Conclusion : quelques conseils utiles. 71
3.5. OPERATIONS SUR UNE RI. 72
3.5.1. Arbres et emplacements d'arbres. 72
3.5.2. Opérations de consultation d'un arbre. 74
3.5.3. Opérations de création et modification d'un arbre. 75
3.5.4. Opérations sur les annotations. 76
3.5.5. Opération d'instanciation par filtrage. 77
3.6. UTILISATION D'UNE RI DANS UN ATELIER. 79
3.6.1. Déroutements à l'intérieur des opérations sur la RI. 79
3.6.2. Description d'un décompilateur. 81
3.6.3. Description d'un analyseur syntaxique. 83
3.7. RI DANS L'ATELIER ADELE. 86
3.7.1. Grammaire abstraite du langage Pascal. 86
3.7.2. Annotations. 87
3.7.3. Implémentation. 89

CHAPITRE 4 : ANALYSE CONTEXTUELLE INCREMENTALE. 93
4.1. LES CHOIX POSSIBLES. 94
4.2. PRESENTATION DU PROBLEME. 96
4.3. UTILISATION DE GRAMMAIRES ATTRIBUEES. 98
4.3.1. Grammaires attribuées. 98
4.3.2. Analyse par propagation des modifications. 99
4.3.3. Analyse par annulation-réévaluation. 101
4.3.4. Analyse "optimale" par propagation des modifications. 102
4.3.5. Critique de la méthode. 103
4.4. RELATIONS DE DEPENDANCE LOINTAINE. 105
4.4.1. Ensembles de relations contextuelles. 105
4.4.2. Evaluation non incrémentale. 107
4.4.3. Evaluation incrémentale. 109
4.4.4. Critique de la méthode. 110
4.5. ANALYSE INCREMENTALE DANS L'ATELIER ADELE. 112
4.5.1. Principe. 112
4.5.2. Algorithmes. 114

CHAPITRE 5 : CONCLUSION. 119

. 5.1. EVALUATION DU PROTOTYPE. 119

5.2. PERSPECTIVES. 123

ANNEXE A1 : COMMANDES DE L'EDITEUR ADELE.

ANNEXE A2 : GRAMMAIRE ABSTRAITE DE PASCAL.

BIBLIOGRAPHIE.

INTRODUCTION

La production de logiciel est une activité dont le coût ne cesse de croître, et la qualité elle-même du logiciel produit est parfois mise en doute. Ces problèmes peuvent être imputés aux systèmes existants d'aide à la conception et à la réalisation d'un produit logiciel. Il paraît donc naturel de s'interroger sur la nature de ces systèmes, et de proposer de nouveaux mécanismes. Le domaine étant très vaste, nous nous sommes consacrés aux problèmes que posent l'écriture et la mise au point d'un programme. Notre étude a abouti au développement d'un prototype dans le cadre d'un projet d'atelier de génie logiciel, Adèle (*).

Notre thèse s'articule sur trois points :

- L'utilisation d'un éditeur syntaxique facilite le processus d'écriture d'un programme.

- La syntaxe abstraite constitue une représentation privilégiée d'un programme au sein d'un atelier.

- L'analyse de la syntaxe contextuelle d'un programme au fur et à mesure de son évolution (analyse incrémentale) contribue au confort et à la productivité des programmeurs.

Une idée sous-jacente qui constitue un fil directeur à nos propos, est l'importance de l'interface entre l'usager et l'application pour améliorer la production de logiciels.

Nous développons ces propositions selon le plan suivant :

Le premier chapitre constitue une introduction détaillée à notre thèse. Il précise la notion d'atelier de génie logiciel et présente l'atelier Adèle comme cadre de notre travail.

Le chapitre deux est consacré aux spécifications externes d'un éditeur syntaxique. Après avoir dégagé l'intérêt d'un tel outil, nous illustrons cette notion par l'exposé des systèmes Mentor et CPS, puis nous détaillons quelques points importants. Nous présentons enfin l'éditeur Adèle.

Le chapitre trois développe la notion de représentation d'un programme au sein d'un atelier. Nous détaillons tout d'abord les raisons qui nous ont poussé à choisir l'arbre de la syntaxe abstraite, puis nous précisons cette notion à l'aide du formalisme des ramifications. Nous indiquons alors comment décrire certains outils de manipulation de programmes de façon paramétrée par le langage et sa syntaxe abstraite. Nous détaillons enfin les principes de la représentation de programmes dans l'atelier Adèle.

Le chapitre quatre est consacré à l'analyse contextuelle incrémentale. Nous présentons deux modèles permettant une approche à ce problème : l'utilisation de grammaires attribuées et la gestion des relations de dépendance lointaine. Nous montrons enfin comment cette dernière approche a été utilisée avec succès dans l'atelier Adèle.

La conclusion de cette thèse consiste tout d'abord en une évaluation du prototype réalisé, puis nous soulignons les limites de notre travail et les perspectives de nouvelles études.

Edition syntaxique, représentation de programmes et analyse incrémentale sont trois notions intimement corrélées, parfois difficiles à présenter séparément. Aussi, nous prions le lecteur de cette thèse de nous excuser s'il est obligé d'en adopter une lecture non linéaire, afin d'obtenir une bonne compréhension globale. Notre plus grand souhait est finalement de faire comprendre les problèmes que nous avons rencontrés, plutôt que d'en présenter des solutions hélas trop partielles.

(*) Le projet Adèle a été partiellement financé par l'ADI et par le CNET de Lannion, dans le cadre du projet Concerto.

CHAPITRE I

ATELIERS DE GENIE LOGICIEL

I.1. INTRODUCTION.

La production de logiciel est une activité en très forte expansion depuis plusieurs années. Pourtant, d'un point de vue économique, on parle depuis toujours de crise du logiciel ; force est de constater que les coûts de production ne cessent de croître, et que la qualité des produits est souvent médiocre.

L'augmentation des coûts de production provient de la taille et de la complexité croissantes des logiciels. Les possibilités actuelles des matériels, tant en puissance de calcul qu'en taille de mémoire, permettent de réaliser des logiciels techniquement inconcevables il y a seulement quelques années. Par exemple, dans le domaine des télécommunications, le logiciel de gestion d'un autocommutateur représente entre cinq cent mille et un million de lignes de code source <André 82>. Les concepteurs d'un logiciel n'en maîtrisent que difficilement la complexité et la réalisation, et il en résulte fréquemment des surcoûts et des retards de production. Dans les secteurs où la concurrence internationale est vive, tout retard de livraison dû aux difficultés de mise au point d'un produit aboutit généralement à la perte d'un marché.

La qualité des logiciels est souvent remise en cause : l'utilisateur d'un logiciel est obligé de s'adapter au produit, alors que c'est l'inverse qui devrait se produire. Enfin, l'accroissement de la durée de vie d'un logiciel met en évidence les difficultés rencontrées pour comprendre son fonctionnement et pour le faire évoluer.

L'analyse de ces problèmes montre la nécessité de définir et d'élaborer de véritables environnements d'aide à la production de logiciel, ne se contentant plus de proposer une panoplie d'outils disparates et indépendants, mais offrant au contraire un ensemble homogène et cohérent d'outils partageant la même finalité : l'aide au développement d'un logiciel. De tels environnements sont nommés de manière désormais classique, ateliers de développement de logiciel, ou ateliers de génie logiciel.

Il est hors de notre propos de développer tous les services offerts par un atelier de génie logiciel : nous nous sommes bornés à étudier les problèmes d'écriture et de mise au point d'un composant du produit final. Mais nous voulons préciser deux fonctions d'un atelier qui nous semblent fondamentales : l'archivage de composants et l'interface homme/machine.

La décomposition fonctionnelle d'un logiciel s'exprime aisément par la notion classique de module. Un module est généralement la donnée d'une interface et d'une réalisation, ou corps. Cette notion est offerte par de nombreux langages, comme Mesa, Ada et Modula-2. Mais peu de systèmes assurent un contrôle des différentes réalisations d'un même module (ses versions). L'intégration de composants pour aboutir au produit final est alors une opération difficile et périlleuse : l'oubli d'un composant ou l'utilisation de composants incompatibles sont des erreurs fréquentes, parfois difficilement détectables, toujours coûteuses. Un atelier de génie logiciel doit donc proposer un service d'archivage, connaissant la notion de module et tenant compte des relations entre modules.

Le dialogue entre un être humain et un outil informatique fait dorénavant appel à des techniques et du matériel sophistiqués : l'utilisation de terminaux graphiques à haute résolution et de dispositifs de "désignation directe" offre un certain niveau de confort et, par là-même, une meilleure productivité. Mais les concepteurs d'un outil ne doivent pas limiter leur effort à ce niveau matériel, car leur tendance naturelle est de construire un dialogue "sur mesure" : les outils proposent souvent des interfaces usager/application très différentes, même s'ils offrent des fonctions voisines. En conséquence, l'utilisateur doit s'adapter continuellement à l'outil dont il se sert, et mémoriser différents modes de dialogue. Un atelier doit donc proposer une interface usager/application conçue sur un modèle unique, afin d'offrir un dialogue homogène.

1.2. AIDE A L'ECRITURE D'UN COMPOSANT LOGICIEL.

1.2.1. Les limites du schéma classique d'écriture.

L'activité d'un programmeur peut se décomposer dans ce qu'on appelle le cycle d'écriture d'un programme :

- construction et modification du programme à l'aide d'un éditeur de textes. La documentation interne au module est généralement réalisée simultanément (fort heureusement, tous les langages admettent la notion de commentaire).

- compilation du programme, c'est-à-dire vérification de sa conformité au langage de développement, et traduction dans un langage proche des langages machine. Cette phase est souvent précédée par l'utilisation d'un préprocesseur effectuant des transformations textuelles : mise en page, utilisation de macro-commandes, etc.

- édition de liens du programme avec d'autres composants et avec la bibliothèque d'exécution du langage.

- chargement du programme en mémoire et exécution. L'exploitation des résultats peut faire appel à des outils de mise au point adaptés au langage, ou à des techniques plus archaïques ("dump post mortem").

Cette chaîne de production comporte de nombreux inconvénients, dont la lourdeur de sa mise en oeuvre. La succession imposée des commandes est rarement assurée automatiquement. Ceci provoque régulièrement des erreurs qui proviennent le plus souvent d'un manque d'attention de l'utilisateur, et qui imposent de recommencer le cycle. Ces erreurs résultent souvent du changement d'environnement mental que doit effectuer le programmeur lorsqu'il change d'outil.

L'absence de vérifications à l'édition de liens peut aboutir à des erreurs de mise en correspondance des ressources exportées et importées. Ces erreurs, dues à un manque de contrôle de la modularité, sont souvent très difficiles à détecter lors de la phase de mise au point.

Enfin, une erreur est rarement détectée dans la phase où elle prend sa source. Le programmeur hésite alors à reprendre le cycle à son début, cette

opération étant toujours longue. Il a donc tendance à "corriger" l'erreur par des moyens plus ou moins avouables. Par exemple, à la mise au point, il est tenté de supprimer une instruction, de réinitialiser correctement une variable, etc. Ces "corrections" peuvent bien sûr aboutir à des erreurs de "second niveau". Mais surtout, l'utilisateur peut oublier de les retranscrire dans le programme source. Et, même s'il le fait, cette opération peut porter en soi les germes de nouvelles erreurs, plus difficiles à corriger, car le programmeur est alors persuadé de les avoir corrigées, et perd son temps à chercher les causes des erreurs ailleurs qu'au bon endroit.

D'un point de vue interne, on constate que les composants de la chaîne d'écriture imposent leur propre représentation du même objet, le programme, ce qui augmente le volume des objets à archiver :

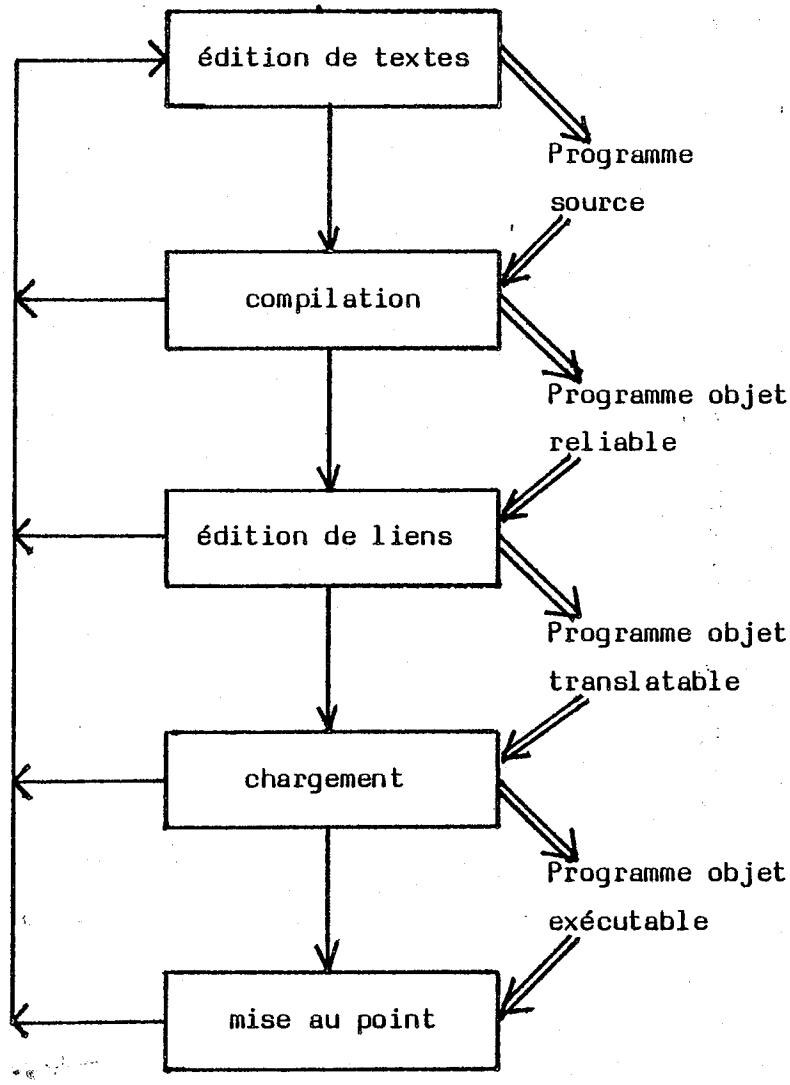


Fig. 1.1. Cycle d'écriture d'un composant.

En résumé, le cycle d'écriture classiquement employé paraît mal adapté aux besoins du programmeur. En particulier, l'édition et la mise au point d'un programme sont effectuées de façon indépendante du langage utilisé.

1.2.2. Réduction du cycle d'écriture.

Diverses solutions ont été proposées pour résoudre les problèmes évoqués. Nous retiendrons deux exemples : le premier concerne un système d'exploitation, Multics, et le second, un langage, Lisp.

Le système Multics propose un mécanisme d'édition de liens et de chargement dynamiques : ces opérations s'effectuent à l'exécution du programme et sont transparentes à l'utilisateur. Il existe de plus un outil de mise au point, Probe, offrant des commandes adaptées au langage de programmation. Cependant, édition et mise au point nécessitent deux représentations distinctes du programme : le texte source et le programme objet (augmenté d'une table de symboles). L'utilisateur n'a donc pas la possibilité de modifier son programme source au moment de la mise au point.

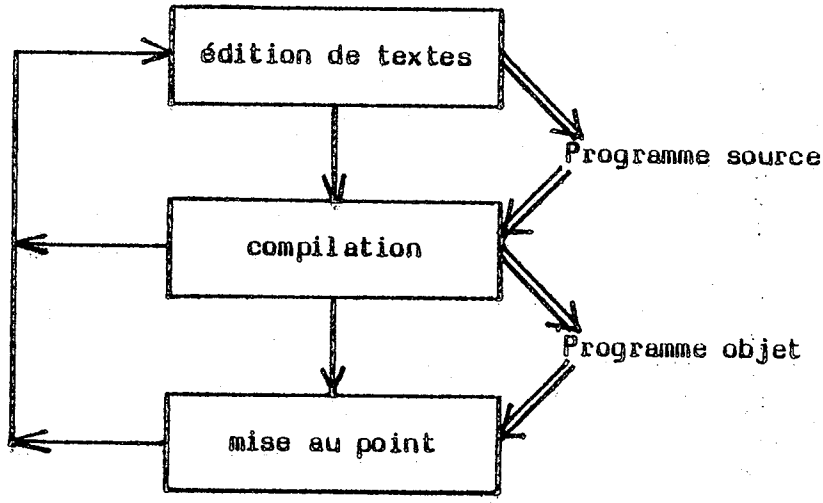


Fig. 1.2. Cycle d'écriture du système Multics.

La commodité d'utilisation de langages tels que Lisp provient pour une bonne part de l'existence d'environnements très élaborés, basés sur une interprétation des programmes écrits dans ces langages : il n'y a pas de phase de compilation, toujours longue et coûteuse dans un système classique (mais elle est possible après la mise au point). Au contraire, dans un système comme Interlisp, édition et mise au point paraissent entremêlées : les répercussions

d'une modification du programme sont immédiates, et l'utilisateur ne change pas d'environnement pour éditer ou mettre au point son programme.

Nous pensons qu'il est possible d'utiliser cette méthode pour tous les langages de programmation, ce qui permet de réduire considérablement le cycle d'écriture : la construction et la modification d'un programme s'effectuent à l'aide d'un éditeur syntaxique, adapté au langage de programmation utilisé. L'exécution plus lente du programme - rendue tolérable par l'évolution du matériel - est largement compensée par les facilités accrues de mise au point.

Pour que l'édition et la mise au point ne soient pas des activités autonomes, mais apparaissent entremêlées, il est nécessaire qu'elles partagent la même représentation du programme (nous l'appellerons par la suite Représentation Interne, ou RI). De cette manière, l'utilisateur peut demander une exécution, l'interrompre, modifier son programme, puis reprendre l'exécution, qui tient immédiatement compte des modifications. Pour l'utilisateur, tout se passe alors comme si éditeur et interpréteur étaient deux composants d'un même outil, s'exécutant en coroutines.

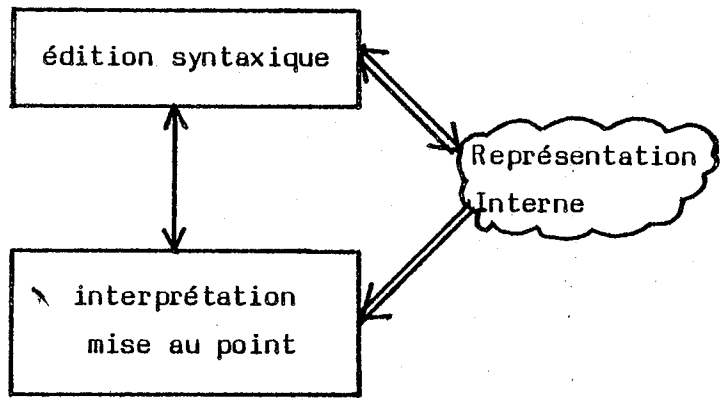


Fig. 1.3. Cycle d'écriture proposé.

Lorsque le programme est jugé correct et que son exploitation peut commencer, il doit être compilé pour obtenir un code objet correspondant à la machine de développement ou à toute autre machine (compilation croisée). Le compilateur ne disparaît donc pas de l'environnement de développement. Par contre, ses objectifs changent : il est moins nécessaire qu'il soit rapide, puisqu'il n'est plus utilisé systématiquement après chaque mise à jour du programme ; mais il doit produire du code d'excellente qualité, optimisé en encombrement mémoire et en vitesse d'exécution.

1.3. PRESENTATION DU PROJET ADELE.

1.3.1. Objectifs.

Adèle (Atelier de DEveloppement de LogicIEl) est un environnement de programmation destiné à une petite communauté de chercheurs. Un projet de recherche peut se caractériser de la manière suivante :

- petite équipe (de 5 à 10 personnes),
- chaque individu participe à toutes les phases du projet,
- applications de taille moyenne (quelques dizaines de milliers de lignes) mais difficiles à concevoir et à mettre au point,
- maintenance quasiment inexistante : les produits sont des maquettes destinées à valider des idées nouvelles.

L'atelier Adèle vise en conséquence à faciliter la construction et la mise au point de programmes. Les phases de spécification et d'analyse du produit ne sont pas gérées par l'atelier.

Les principes retenus lors de la conception d'Adèle ont été les suivants :

- L'atelier propose une méthode de décomposition des produits en modules : le langage de programmation retenu est Pascal, étendu pour la modularité.

- Un système d'archivage permet la gestion des versions d'un composant ainsi que leurs droits d'accès.

- Le cycle d'écriture d'un composant est basé sur un éditeur syntaxique et un interpréteur. Les programmes possèdent une représentation interne unique.

- L'interface usager-outils est rendue uniforme par l'utilisation d'un médiateur pour la définition et l'acquisition des requêtes, et d'un compositeur pour l'affichage des résultats.

- Un introducteur permet l'entrée dans l'atelier de programmes Pascal déjà existants, et un compilateur produit du code objet pour différentes machines cibles. Enfin, un décompilateur permet de reconstituer un programme Pascal standard, permettant le transfert vers d'autres environnements.

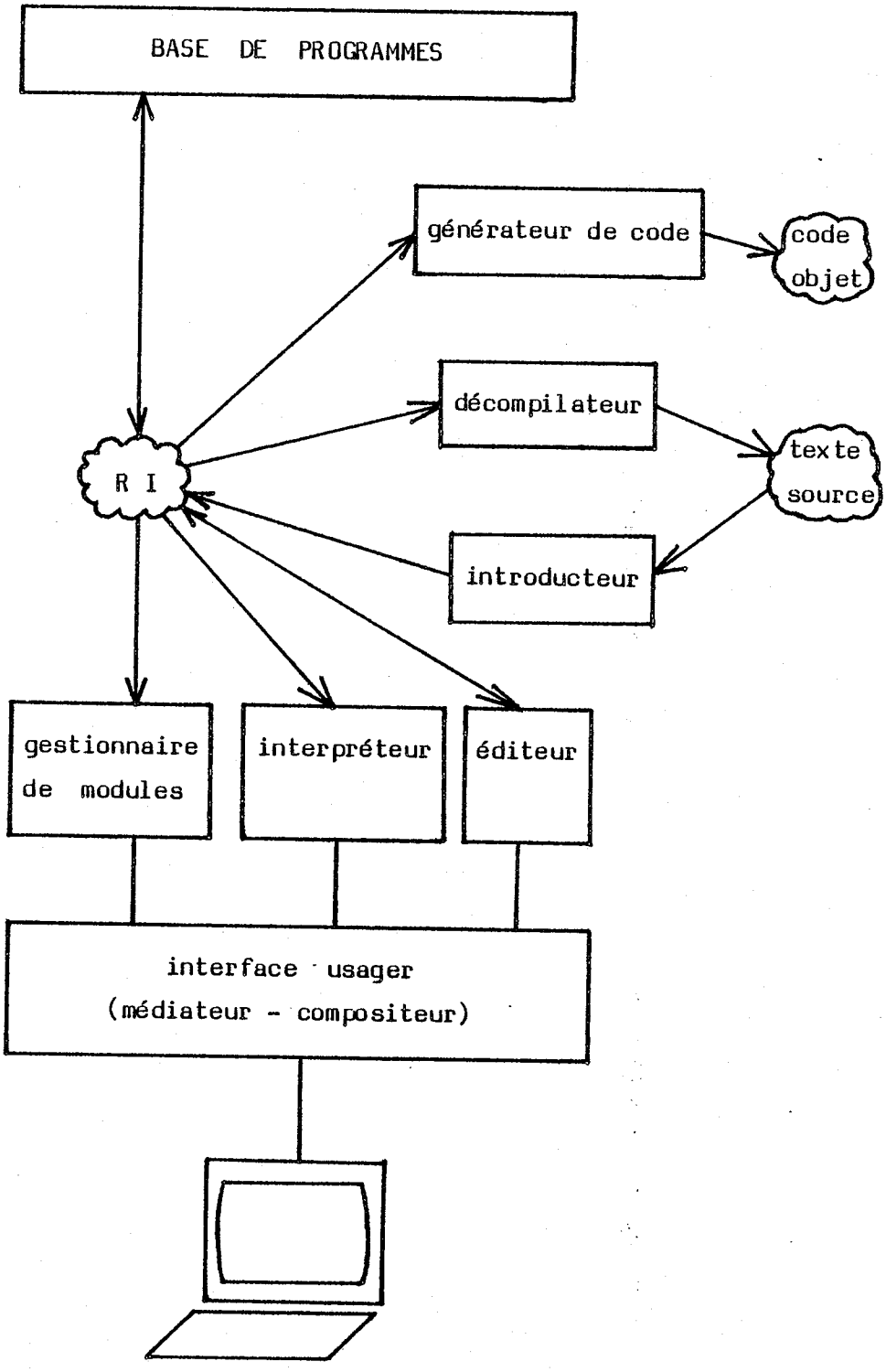


Fig. 1.4. : Structure de l'atelier Adèle.

Le support matériel d'Adèle se compose actuellement de terminaux alphanumériques (munis de quelques possibilités semi-graphiques), connectés en temps partagé sur le système Multics de Grenoble. Il est envisagé par la suite d'utiliser des postes de travail individuels, équipés d'écrans graphiques à haute résolution, et connectés en réseau local.

Les chapitres suivants sont consacrés à l'édition syntaxique et la représentation interne des programmes. Nous nous bornerons donc à développer ici le service d'archivage, l'interpréteur et l'interface usager. Le lecteur intéressé par plus de détails pourra se reporter à la bibliographie Adèle.

1.3.2. Extensions au langage Pascal.

Le principe de modularité retenu dans Adèle reprend les notions classiques d'interface, qui décrit les ressources fournies par un module, et de corps, qui est la réalisation de ces ressources.

Une interface peut contenir les déclarations permises en Pascal : constante, type, variable, procédure et fonction. Un corps utilise implicitement les déclarations de son interface. Il peut aussi utiliser d'autres interfaces, grâce à la clause import (un objet x d'une interface importée I est désigné par la notation x_I).

Il est possible de protéger les objets exportés par un module, à l'aide des clauses ronly, private et limited :

ronly : accès à une variable en lecture seulement ;
private : un type privé est tel que l'on ne peut que déclarer des variables de ce type, leur affecter une valeur, les comparer, et les passer en paramètres de procédure ou de fonction ; il est donc impossible d'accéder directement à ses composants élémentaires ;
limited : un type limité est tel que l'on peut seulement déclarer des variables de ce type et les passer en paramètres.

La dernière extension au langage est la notion d'environ : un environ est une interface ne contenant que des constantes et des types, utilisables dans une interface ou un environ par la clause include (similaire à une inclusion textuelle). Si un environ E1 inclut un environ E2, les objets de E2 sont utilisables comme des déclarations de E1 : la visibilité est transitive.

```
Ex 1 : interface  sgf_réduit ;
        limited fich ;
        private nomfich ;
        ronly nbfich ;
        const  lgnommax = 30 ;
        type  nomfich = packed array [1..lgnommax] of char ;
              fich = record
                        nom : nomfich ;
                        access : (seq, indexé, direct) ;
                        adr : integer ;
                     end ;
        var  nbfich : integer ;
        procedure créerfich (var idfich : fich ; nom : nomfich) ;
        procedure détruirefich (idfich : fich) ;
    end.

    body  base ;
        import  sgf_reduit ;
        var  f1, f2 : sgf_fich ;
             nomf1 : sgf_nomfich ;
        begin
             nomf1 := 'fichier' ;
             sgf_créerfich (f1, nomf1) ;
             f2 := f1 ;                          (1)
             nomf1[1] := 'a' ;                   (2)
             sgf_nbfich := 0                     (3)
        end.
```

Les lignes (1), (2) et (3) sont erronées :
(1) affectation de deux variables de type limité,
(2) accès à un élément d'une variable de type privé,
(3) affectation d'une variable en lecture seule.

Ex 2 : (cet exemple est sémantiquement identique au précédent)

```
environ  nom_réduit ;
     const  lgnommax = 30 ;
     type  nomfich = packed array [1..lgnommax] of char ;
end.

environ  fich_réduit ;
     include nom_réduit ;
     type  fich = record
                      nom : nomfich ;
                      accès : (seq, indexé, direct) ;
                      adr : integer ;
                  end.
end.

interface  sgf_réduit ;
     include  fich_réduit ;
     limited  fich ;
     private  nomfich ;
     ronly  nbfich ;
     var  nbfich : integer ;
     procedure  créerfich (var idfich : fich ; nom : nomfich) ;
     procedure  détruirefich (idfich : fich) ;
end.

body  base ;
     import  sgf_réduit ;
     var  f1, f2 : fich ;
          nomf1 : nomfich ;
     begin
          nomf1 := 'fichier' ;
          sgf_créerfich (f1, nomf1) ;
          f2 := f1 ;                          (1)
          nomf1[1] := 'a' ;                   (2)
          sgf_nbfich := 0 ;                   (3)
     end.
```

1.3.3. Archivage des modules.

La base de programmes permet de gérer les différentes versions d'un module suivant trois notions :

- La **famille** définit un ensemble d'interfaces "voisines", logiquement corrélées. La résignation d'une interface est donnée par le nom composé :
<famille> _ <interface>
(les crochets indiquent que la spécification de l'interface est facultative : le système choisit alors une interface par défaut, dite interface standard).

- La **version** (d'un corps) en définit une réalisation particulière. A chaque interface est associée une version par défaut.

- La **révision** d'une version est le résultat de la modification d'une version. Deux révisions d'une même version doivent importer les mêmes ressources. La révision par défaut est toujours la plus récente.

Le schéma version-révision est devenu classique. A notre connaissance, la notion de famille est propre à Adèle. Le plus souvent, on obtient plusieurs interfaces dans une même famille par restriction de l'une d'entre elles : soit en limitant les accès aux ressources exportées (clauses private, limited et ronly), soit en en diminuant le nombre.

Le nom complet d'un exemplaire de module est le suivant :
<famille> _ <interface> . <version> . <revision>
Dans l'exemple 1 du paragraphe précédent, le corps "base" importe la révision standard de la version standard de l'interface "réduit" de la famille "sgf".

Chaque composant de la base de programmes possède un **manuel**, qui indique son mode d'emploi. Un manuel est composé de rubriques précisant des attributs et des contraintes :

- Les attributs peuvent préciser n'importe quelle propriété définie par les concepteurs. Certains sont gérés par le système, comme la date et l'état du composant. D'autres précisent la visibilité du composant, c'est-à-dire quelles sont les familles qui pourront l'utiliser.
- Les contraintes expriment les sélections imposées : l'utilisation d'une version d'un module peut forcer le choix d'une version d'un autre module (implication) ou l'interdire (exclusion).

Signalons enfin la possibilité de "détypage" d'objets exportés d'une même famille, permettant l'utilisation du même objet par des types différents : l'expérience d'écriture de logiciels importants montre en effet que le typage rigide de Pascal est une contrainte qu'il est (malheureusement !) parfois nécessaire de lever. Les solutions habituellement retenues pour effectuer ce détypage (variantes d'article, absence de contrôle à l'édition de liens) ne montrent pas explicitement la transformation de type, et peuvent aboutir à des erreurs de programmation et à des problèmes de portabilité. C'est pourquoi la solution retenue dans Adèle rend les détypages explicites, par leur mention dans le manuel associé à une interface.

Ex :

```
interface F_I ;
    type ptr = ↑enreg ;
        enreg = record ... end ;
    function f (p : ptr) : ptr ;
end.

interface F_Irestreinte ;
    type noeud = ↑integer ;
    function f(n : noeud) : noeud ;
end.

manual F_Irestreinte ;
    rest --> I (noeud = ptr) ;          (1)
end.
```

(1) indique que F_Irestreinte est une restriction de F_I et établit l'identité des deux types noeud et ptr.

1.3.4. Interpréteur.

L'interpréteur-metteur au point peut être activé à tout moment lors de l'édition d'un programme. Il s'exécute jusqu'à la rencontre d'une condition d'arrêt :

- fin du programme,
- instruction incomplète ou non conforme au langage,
- détection d'une erreur d'exécution,
- rencontre d'un point d'arrêt,
- fin d'une instruction élémentaire en mode pas à pas,
- appel utilisateur.

Les principales commandes de l'interpréteur sont les suivantes :

- Mise en mode pas à pas : le contrôle est rendu à l'utilisateur après l'exécution de chaque instruction.

- Mise en mode exécution lente : une temporisation est introduite pour permettre l'exécution d'une instruction en une seconde environ.

- Mise en mode exécution normale : l'interprétation se déroule sans temporisation, l'utilisateur peut l'arrêter à tout moment.

- Evaluation d'expressions : si l'exécution est arrêtée sur une instruction possédant une expression arithmétique ou logique, l'utilisateur peut en demander l'évaluation sans traiter l'instruction complètement. Il est ainsi possible de savoir si la condition d'arrêt d'une instruction répétitive est respectée.

- Visualisation de variables : l'interpréteur indique la valeur de la variable inspectée, ainsi que la dernière instruction dans laquelle la variable a été affectée.

- Modification de variables : l'utilisateur peut modifier à tout moment la valeur d'une variable. Les demandes ultérieures de visualisation indiquent alors : "variable affectée au clavier".

- Trace de variables : l'interpréteur affiche la valeur d'une variable tracée à chaque fois qu'elle est modifiée.

- Protection d'une variable : l'interpréteur s'arrête à la rencontre d'une instruction modifiant la variable protégée.

- Points d'arrêts : l'interpréteur s'arrête après avoir exécuté n fois l'instruction possédant un point d'arrêt, où n est un entier optionnel fourni à la pose du point d'arrêt (par défaut, n = 0).

- Visualisation de la pile d'exécution : l'interpréteur affiche l'ensemble des procédures et fonctions actives.

L'interpréteur utilise quatre fenêtres pour l'affichage des résultats :
(1) - le programme (l'instruction en cours est mise en évidence),
(2) - les entrées/sorties du programme,
(3) - la trace des variables,
(4) - les commandes et messages d'erreurs.

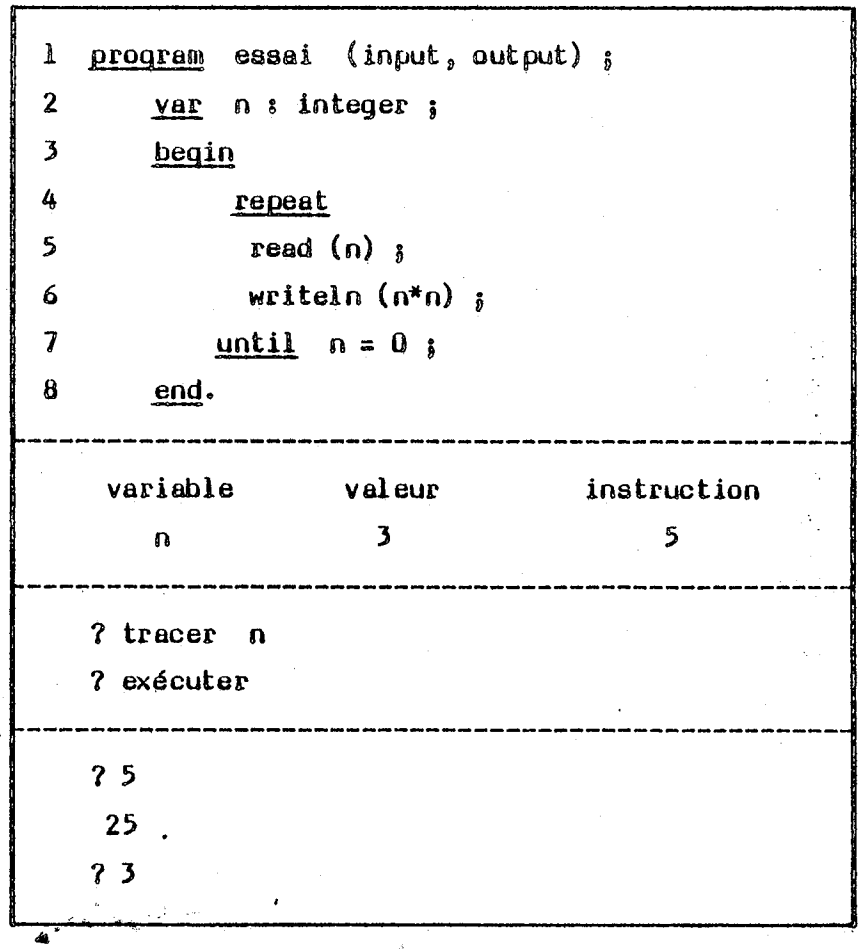


Fig. 1.5. Les fenêtres de l'interpréteur.

1.3.5. Interface usager-application.

Les outils de l'atelier dialoguent avec l'utilisateur par le biais du compositeur et du médiateur : le compositeur gère une structure arborescente d'affichage, l'arbre de boîtes, tandis que le médiateur prend en charge les commandes de l'utilisateur, et détermine l'outil concerné (les outils de l'atelier sont tous potentiellement actifs à tout instant).

Les interactions transitent par deux autres composants, qui définissent les notions d'appareil logique et d'appareil virtuel.

A chaque application de l'atelier correspond un, et un seul, **appareil virtuel**, la libérant de tout problème de compétition d'accès au terminal logique. L'appareil virtuel utilise la notion classique de fenêtre, définie comme une portion de l'appareil logique. A tout instant, une application possède un certain nombre de fenêtres, visibles ou invisibles, et en est l'unique propriétaire. La donnée d'une fenêtre est donc suffisante pour déterminer le destinataire d'une commande.

La notion d'**appareil logique** permet de masquer les limitations des terminaux physiques. L'appareil logique de l'atelier est composé actuellement de :

- Un écran alphanumérique possédant quelques possibilités semi-graphiques : surbrillance, clignotement, etc.

- Un organe de désignation : curseur ou souris.

- Un clavier composé de trois parties distinctes : touches alphanumériques, touches de désignation d'un objet et touches de désignation d'une commande. Cette partition permet une distinction nette entre texte, objet et opérateur. En pratique, la distinction peut s'effectuer en utilisant des touches spéciales du terminal physique : "escape", "control", etc.

Le schéma de l'interface usager est donc le suivant :

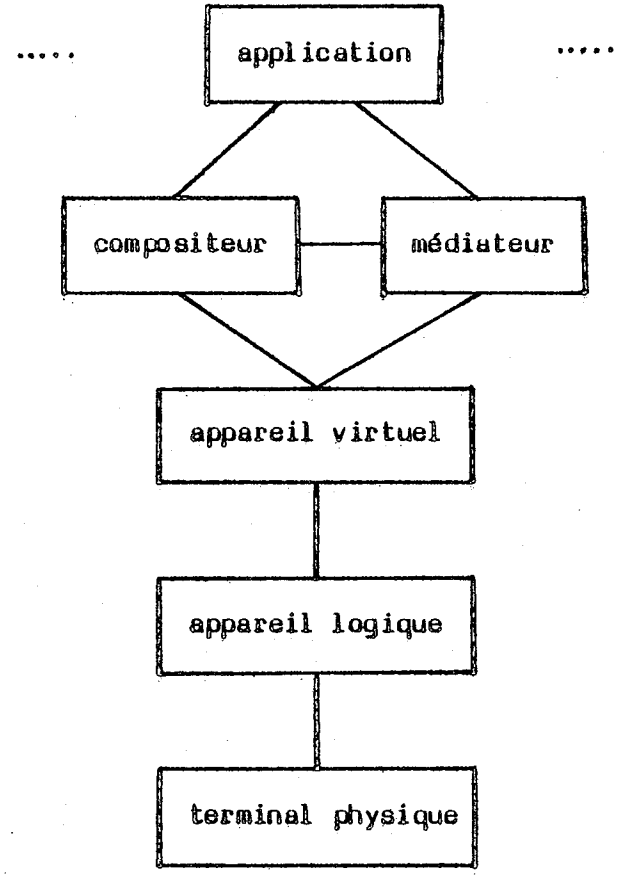


Fig. 1.6. Interface usager de l'atelier Adèle.

1.3.6. Compositeur.

Le compositeur a pour but de décharger les applications de tout problème de visualisation des objets qu'elles manipulent, en fournissant la notion de boîte. En entrée, une boîte définit une unité d'information désignable à l'aide des dispositifs de sélection. En sortie, elle exprime des relations entre les informations à afficher, et permet une spécification dynamique d'attributs visuels.

La structure arborescente permet un mécanisme d'héritage des attributs : les attributs d'une boîte sont propagés à toutes ses descendantes qui ne le possèdent pas.

Les attributs de composition définissent l'organisation des boîtes dans une fenêtre :

- la composition horizontale, H, impose un alignement horizontal des boîtes,
- la composition verticale, V, impose un alignement vertical,
- H ou V indique une composition horizontale si possible, ou sinon verticale,
- H et V indique un remplissage des lignes horizontales autant que possible, avec passages à la ligne si nécessaire : mais une boîte fille ne doit pas se trouver à cheval sur deux lignes.

Les attributs visuels permettent de spécifier la fonte et la couleur des caractères et du fond, ainsi que les effets spéciaux : clignotement, surbrillance, etc. Enfin, l'attribut de corrélation permet à l'application de retrouver à quel objet correspond un sous-arbre de boîtes.

L'exemple suivant montre l'arbre de boîtes correspondant à un programme Pascal, et présente deux affichages distincts, suivant la taille de la fenêtre support : while x > y do x := x - y

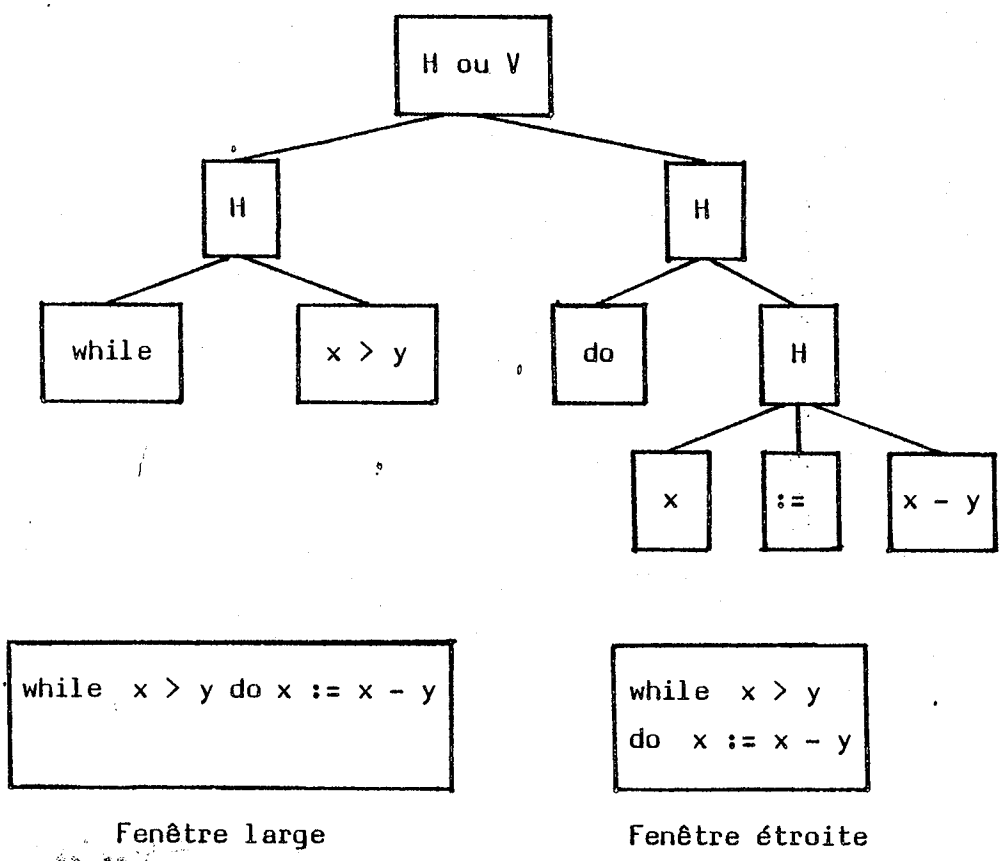


Fig. 1.7. Affichage de l'arbre de boîtes.

Le concepteur d'une application doit réaliser un compromis entre la "taille" d'une boîte feuille et celle de l'arbre des boîtes. Ici, tout caractère désigné dans le texte "x-y" représente le même objet de l'application : il n'est pas possible de désigner plus finement une partie de l'expression. Cela aurait été possible en définissant l'arbre comme une composition horizontale des boîtes feuilles "x", "-", et "y". Mais l'arbre complet du programme serait alors beaucoup plus important.

L'affichage est incrémental : dès que l'application modifie une partie de l'arbre, le compositeur calcule la nouvelle image à visualiser, en évitant bien sûr d'effacer l'écran et de tout réafficher.

1.3.7. Médiateur.

Le médiateur définit un modèle d'interaction entre l'application et l'usager, dont les grands principes sont :

- Emploi d'opérateurs universels : par exemple, l'action de destruction d'un objet porte toujours le nom "détruire", quelle que soit l'application concernée. Le nombre de ces opérateurs est volontairement limité pour en faciliter l'apprentissage. Par exemple, l'action d'imprimer un programme consiste à le copier dans l'objet imprimante.

- Redondance de la désignation : un objet, comme un opérateur, peut être désigné de trois manières : sélection d'une représentation visuelle, frappe de son nom, ou emploi d'une touche "synonyme" accélérant le processus de la frappe. L'utilisateur peut ainsi choisir à tout moment le mode de désignation qui lui convient le mieux.

- Gestion des fenêtres : le médiateur contrôle l'organisation de l'écran logique. Les applications lui demandent la création et la destruction d'une fenêtre, et il en détermine la localisation et la taille initiale suivant la taille de l'écran. De plus, l'utilisateur peut à tout moment modifier les choix du médiateur.

- Possibilité de modifier un objet ou un opérateur : le médiateur conserve une liste des propriétés associées à chaque objet et opérateur. Pour un opérateur, cette liste indique le mode d'interaction : novice ou expert, avec ou sans confirmation, avec ou sans création de fenêtre, etc. A un type d'objet correspond la liste des opérateurs qui lui sont applicables. L'utilisateur peut ainsi décider à tout moment que la destruction d'un objet "programme" nécessite une confirmation, alors que celle d'un objet "instruction" n'en nécessite pas. De même, le mode novice entraîne l'utilisation d'un formulaire guidant l'usager pas à pas, tandis que le mode expert limite le dialogue au strict minimum.

Le médiateur apparaît donc comme un gestionnaire d'applications. Il propose un modèle d'interaction homogène, évolutive et dirigée par l'utilisateur.

Cette approche ne prétend pas résoudre tous les problèmes : encore faut-il que les concepts proposés par l'application soient les bons. Un système mal conçu ne peut pas être caché à l'utilisateur, même par une interface usager magistralement imaginée !

CHAPITRE II

PRINCIPES DE L'EDITION SYNTAXIQUE

2.1. PRESENTATION GENERALE.

La plupart des éditeurs de textes se limitent à manipuler des lignes composées de caractères. Ces notions ne paraissent pas toujours les plus naturelles pour manipuler un programme. Au contraire, un programmeur utilise intuitivement la notion d'unité syntaxique ; pour s'en convaincre, il suffit d'observer quelqu'un penché sur une liste de compilation, occupé à tracer des traits verticaux pour faire ressortir la structure de son programme...

Un éditeur syntaxique permet à l'utilisateur de manipuler des unités syntaxiques. Par exemple, un éditeur Pascal fournira une commande d'insertion de l'unité "tantque" dans un programme ; l'utilisateur verra alors sur son écran :

```
while (* expression *)
do    (* instruction *)
```

Cette unité "tantque" est composée de deux emplacements, correspondant respectivement à la condition d'arrêt de la boucle et à l'instruction à itérer. L'utilisateur peut remplir un emplacement par une nouvelle unité syntaxique, elle-même composée d'emplacements, ..., jusqu'à ce qu'il aboutisse à un programme complet.

L'utilisateur construit son programme à l'aide d'entités significatives, et ne se préoccupe plus de leur représentation concrète : l'éditeur introduit les mots-clés nécessaires, et se charge de la présentation du programme ; il décale l'affichage des instructions en fonction de leur degré d'imbrication, il insère automatiquement les séparateurs lexicaux (comme le point-virgule séparant deux instructions), etc.

L'utilisation d'un éditeur syntaxique permet de simplifier la tâche d'écriture d'un programme :

- Il devient impossible d'écrire des programmes ne respectant pas la syntaxe hors contexte d'un langage. Avec un éditeur de textes, il arrive d'oublier,

de mal utiliser ou de mal orthographier un mot-clé ou un séparateur lexical du langage. Un compilateur rencontrant, par exemple, une structure mal "parenthésée", a généralement un mauvais comportement : il n'est pas rare de voir s'imprimer plusieurs pages de messages d'erreurs, dont seul le premier est réellement significatif. Le programmeur doit alors localiser la source de l'erreur, puis la corriger, avant de pouvoir relancer une compilation. On voit donc tout l'intérêt d'éviter à leur source de telles erreurs.

- La présentation du programme est normalisée au moment de sa création : cela évite à l'utilisateur la frappe de caractères d'espacement, ou l'appel ultérieur d'outils de mise en page.

- Les commentaires sont généralement considérés comme des "attributs" d'une unité syntaxique : le problème de savoir ce que commente un commentaire situé entre deux instructions n'existe donc plus pour le lecteur d'un programme créé par un éditeur syntaxique.

- L'équivalent de la fonction de recherche d'un éditeur de textes peut être défini dans un éditeur syntaxique, qui offre ainsi des primitives puissantes de transformation d'un programme, tout en respectant les notions du langage. Par exemple, la modification de toutes les occurrences de l'identificateur "gin" par l'identificateur "rhum" ne remplace pas le mot-clé "begin" par "berhum", comme ce serait le cas avec un éditeur de textes ! Un autre exemple est la facilité d'insertion d'une instruction de trace en tête de toutes les procédures d'un programme.

L'utilisation d'un éditeur syntaxique permet donc de réduire les risques d'erreurs et accélère l'écriture d'un programme. Elle n'est cependant pas exempte de difficultés : l'exemple des expressions prouve que leur introduction préfixée choque par trop les habitudes, et qu'une écriture textuelle est préférable dans ce cas. Ceci montre bien l'importance du soin qu'il faut apporter à la conception des fonctions externes d'un éditeur syntaxique, pour que ses utilisateurs ressentent un réel avantage à l'employer à la place d'un éditeur de textes.

C'est pourquoi la suite de ce chapitre est consacrée à l'exposé de deux éditeurs syntaxiques, Mentor et CPS, afin de dégager les principales fonctions d'un éditeur syntaxique, et de bien montrer l'importance de l'interface usager-application.

Le système Mentor <Donzeau-Gouge 75,80> peut être considéré comme la première tentative réussie dans le domaine de l'édition syntaxique, et est à notre avis l'éditeur le plus puissant. Mais son principal défaut est son interface usager.

CPS, "Cornell Program Synthesizer" <Teitelbaum 80,81> propose le schéma édition syntaxique/interprétation comme cycle d'écriture d'un programme, et est agréable à utiliser grâce à une interface usager de bonne qualité.

Ces deux systèmes nous semblent représentatifs de l'état de l'art dans le domaine de l'édition syntaxique. Il convient cependant de rendre hommage aux concepteurs de quelques autres systèmes :

<Griffiths 68> a réalisé un éditeur-interpréteur de programmes PL/1, basé sur un analyseur syntaxique incrémental.

Le système Emily <Hansen 71> propose un éditeur dirigé par la syntaxe concrète du langage.

Legos <Bouchenez 80> est un système dérivé de Mentor, dédié aux petites machines, et intégrant la notion de modularité.

Cona et Copas <Atkinson 78,81> sont des éditeurs-interpréteurs de petits programmes Algol et Pascal.

<Massie 80> a réalisé un éditeur pour le langage extensible Let.

Enfin, Gandalf <Habermann 79>, <Gandalf 82> est un générateur d'environnements de programmation, comprenant un éditeur syntaxique, Aloe ("A Language Oriented Editor") et un système d'archivage de modules.

2.2. MENTOR.

Le système Mentor a tout d'abord été uniquement un éditeur syntaxique pour le langage Pascal, avant de devenir paramétré par le langage de programmation <Mélèse 82>, <Donzeau-Gouge 83>. Il est destiné à être utilisé dans un environnement classique d'écriture de programmes : son résultat est donc un texte source qui alimente les compilateurs du système hôte.

L'utilisateur de Mentor doit connaître la structure sous-jacente à un programme, sa syntaxe abstraite, telle que nous la définissons au chapitre 3 : le langage d'édition, Mentol, est en effet un langage de manipulation d'arbres. L'utilisateur désigne l'endroit courant d'édition dans les termes de père (unité englobante), de fils (unité englobée) ou de frère (unité suivante ou précédente).

Les commandes de Mentol servent essentiellement à modifier un programme déjà existant ; la construction la plus naturelle d'un programme ou d'un fragment de programme consiste en son introduction sous forme textuelle. Le système effectue alors une analyse syntaxique du texte inséré pour le mettre sous forme arborescente : Mentor est donc plus adapté à la modification qu'à la construction d'un programme.

Tout fragment de programme peut être repéré par un marqueur. Le système reconnaît un certain nombre de marqueurs prédéfinis : aK est le marqueur courant d'édition, et est le plus souvent implicite dans les commandes Mentol. aTOP désigne l'ensemble du programme. Enfin, toutes les unités du langage sont repérées par des marqueurs prédéfinis. Par exemple, le marqueur aIF désigne l'instruction conditionnelle :

if $exp then $statl else $stat2

$exp, $statl et $stat2 sont appelées métavariables, correspondant à des emplacements non remplis d'une unité. Elles sont surtout utiles au mécanisme de recherche et instanciation, développé plus loin.

Le système n'assure pas l'affichage du programme sur l'écran : c'est à l'utilisateur de demander l'impression explicitement, grâce à la commande P (print) :

P * affiche l'endroit courant, aK, complètement,
P n (n entier) affiche aK à un certain niveau de détails
(par défaut, n = 5).

La mise en page est assurée par le système.

Ex : (dans tous les exemples, les lignes écrites par l'utilisateur sont précédées d'un point d'interrogation, et les commentaires sont entre parenthèses).

```
? P
if X > 0 then  P(X, [A Y,Z])
else  begin
      Y := Y * 2 ;
      X := 0
      end
? P2
if ## then ## else ...
? P3
if X > 0 then P(X, ##)
else  begin
      ## ; ##
      end
```

L'utilisateur peut positionner un marqueur sur n'importe quel fragment de programme, à l'aide de primitives de navigation dans l'arbre : il est ainsi possible de désigner les père (U), fils (S) et frère gauche (L) et droit (R) d'une unité syntaxique. Il est possible de combiner entre eux ces déplacements.

Ex :

```
? S2 ; P          (aller au deuxième fils de aK)
P(X, A[Y,Z])
? S2 ; P
X, A[Y,Z]
? S-2 ; P         (aller au deuxième fils en comptant de droite à gauche)
X
? R ; P
A[Y,Z]
? U *             (remonter au père tant que c'est possible)
? S3 S-1 ; P      (aller au dernier fils du troisième fils de aK)
X := 0
```

Il est possible de demander le positionnement par recherche d'une unité syntaxique (F :·find).

Ex :

```
? U *
? F aCALL ; P   (aCALL est le marqueur prédéfini de l'appel de procédure)
P(X, A [Y,Z])
? F aIDENT ; P  (aIDENT représente les identificateurs)
P
? .up  < aCALL > ; P  (remonter à un appel de procédure)
P(X, A [Y,Z])
```

La commande .up n'est pas une commande de base de Mentol, mais elle s'écrit facilement en Mentol. De façon générale, l'utilisateur peut créer de nouvelles commandes, faisant appel aux commandes de base de Mentol. Ce mécanisme très souple a été utilisé avec succès pour créer un environnement élaboré autour du langage Pascal.

L'utilisateur peut modifier son programme à l'aide des commandes C (changer le marqueur), I (insérer après l'endroit désigné) et J (insérer avant). L'unité insérée peut être désignée par un marqueur (elle est alors recopiée et non déplacée), ou être rentrée sous forme textuelle.

Ex :

```
? U * ; F aASS ; P      (recherche d'une affectation)
Y := Y + 2
? C aIF ; P
if  $exp  then  $statl  else  $stat2
(Un des grands défauts de Mentor est que l'endroit courant n'est pas
automatiquement positionné sur le premier emplacement vide de l'unité qui
vient d'être insérée : l'utilisateur doit donc constamment effectuer des
commandes de déplacement).
? S2 ; P
$statl
? C &              (introduction textuelle)
[STAT] : Z := Z + 1      (Mentor indique qu'il attend une instruction)
? U ; P
if  $exp  then  Z := Z + 1 else  $stat2
? S2 X S3 ; P      (échange de deux unités)
if  $exp  then  $stat2  else  Z := Z + 1
```

La commande E (évaluer) est la plus puissante de toutes ; associée à la commande de recherche, elle fournit un mécanisme de réécriture : le système identifie les métavariables du modèle de recherche à celles du modèle d'évaluation. Les unités syntaxiques que la recherche associe aux métavariables sont donc récupérées par l'évaluation.

Ex : (transformation d'une instruction répéter en tantque)
(on suppose que l'utilisateur a créé par ailleurs aTQ et aREP)

```
? aREP P
repeat $stat until $cond
? aTQ P
while not ($cond)
do begin
    $stat
    end
? F aREP ; P
repeat
   X := X + Y
   Y := Y + 1
until X > Y
? C E aTQ ; P
while not (X > Y)
do begin
    X := X + Y
    Y := Y + 1
    end
```

L'utilisateur peut ainsi envisager toutes sortes de transformations de son programme, comme par exemple insérer une instruction de trace en tête de toutes les procédures.

Les commentaires sont traités comme des "décorations" d'une unité syntaxique ; ils sont créés et modifiés par la commande Bn, où n est un entier désignant la décoration choisie. La forme la plus générale d'un commentaire est un arbre, c'est-à-dire une structure possédant sa propre syntaxe, et donc manipulable en Mentol. Une forme généralement employée est une suite de lignes, mais il est possible, par exemple, de commenter une instruction Pascal par une autre instruction Pascal (représentant peut-être une autre version), elle-même également commentée, etc. La commande Bn permet l'accès à une décoration, et la commande U0 permet de revenir à la structure commentée.

Mentol possède enfin des instructions de contrôle : des instructions de test, de choix, d'itération et d'appel de procédures Mentol, permettent l'écriture de véritables programmes de manipulation d'arbres. L'environnement Pascal a été conçu de cette manière, et fournit des commandes élaborées : elles permettent ainsi des déplacements plus adaptés au langage ; certaines d'entre elles fournissent un embryon de vérification de la syntaxe contextuelle d'un programme Pascal.

Le principal défaut du système est sa syntaxe rébarbative et, de façon générale, son interface usager. L'utilisateur ne "voit" pas où il en est, et doit demander explicitement l'affichage de son programme. L'insertion d'unités prédéfinies se révèle en pratique inexploitable, car elle oblige l'utilisateur à demander un positionnement avant de "remplir" l'unité insérée. L'usage du mode textuel se révèle donc plus pratique ; mais dans ce cas, l'utilisateur n'est pas exempt d'erreurs syntaxiques, qui peuvent aboutir au rejet d'une partie du texte. Enfin, Mentol est un langage interprété, ce qui rend son utilisation coûteuse.

Néanmoins, l'impression la plus forte reste la puissance de ses fonctions, permettant la mise en oeuvre de transformations complexes d'un programme. Mentor a ainsi été utilisé pour le transport de programmes Pascal <Guerrier 80> entre environnements possédant des extensions du langage peu compatibles (principalement les facilités de compilation séparée) : une fois la procédure Mentol mise au point, la transformation des programmes a pu s'effectuer sans erreur.

Signalons enfin que Mentor, écrit en Pascal, est maintenu sous Mentor, ce qui fait une validation sérieuse des possibilités du système.

2.3. CPS.

CPS fournit à ses utilisateurs un environnement intégré d'édition et de mise au point de programmes PL/CS (sous-ensemble de PL/1, destiné à l'enseignement). Ce système est actuellement utilisé par les étudiants de l'Université de Cornell, New York.

L'opération de base de l'éditeur consiste à remplir les emplacements vides d'une unité syntaxique par d'autres unités du langage. L'éditeur est "orienté écran" : toute opération s'effectue sur les unités visibles et a pour effet de modifier l'affichage du programme. Contrairement à Mentor, la structure de données utilisée par l'éditeur pour représenter le programme est donc transparente à l'utilisateur.

L'emplacement courant d'édition est mis en évidence par le clignotement du premier caractère qui le représente.

Ex :

```
abs : PROCEDURE OPTIONS (MAIN) ;
      DECLARE (list-of-variables) FIXED ;
      {statement}
END   abs ;
```

"list-of-variables" et "statement" sont des emplacements non remplis. Les accolades entourant "statement" indiquent la possibilité d'insérer un nombre quelconque d'instructions.

La frappe de la commande ".i" provoque l'insertion d'une instruction conditionnelle. L'endroit courant devient l'emplacement de la condition, qui est le premier emplacement vide de l'unité insérée. L'utilisateur voit donc le curseur positionné à l'endroit où il désire insérer de nouvelles unités, ce qui lui évite des commandes fastidieuses pour déplacer le curseur.

```
abs : PROCEDURE  OPTIONS (MAIN) ;
      DECLARE  (list-of-variables) FIXED ;
      IF (condition)
      THEN  statement
      ELSE  statement
END abs ;
```

Les instructions sont insérées à l'aide de commandes et les expressions directement dans un mode textuel. Dans notre exemple, la frappe de "sin(x)>0" provoque l'affichage :

```
abs : PROCEDURE  OPTIONS (MAIN) ;
      DECLARE (list-of-variables) FIXED ;
      IF (sin ([x]) > 0)
      THEN  statement
      ELSE  statement
END  abs ;
----- ----- ----- ----- ----- ----- -----
error : undeclared variable
```

L'éditeur a donc effectué une analyse syntaxique de l'expression et détecté une erreur sur la variable x. Le curseur est positionné à l'endroit où l'erreur est détectée, invitant ainsi l'utilisateur à la corriger. Cette correction n'est pas obligatoire : l'utilisateur peut positionner le curseur partout ailleurs. Dans notre exemple, la variable x restera en surbrillance jusqu'à ce que l'utilisateur la déclare, peut-être dans une session d'édition ultérieure.

L'utilisateur dispose de commandes pour déplacer le curseur, c'est-à-dire pour positionner l'endroit courant d'édition. C'est surtout sur ce point que CPS diffère de Mentor : ces déplacements ne se réfèrent pas à l'arbre sous-jacent mais à la structure textuelle du programme, considéré comme une suite linéaire d'emplacements ; "down" déplace le curseur sur le premier caractère de l'emplacement suivant, et "up" sur l'emplacement précédent. Dans l'exemple qui suit, nous avons souligné les caractères où peut être positionné le curseur avec ces deux commandes :

```
abs : PROCEDURE  OPTIONS (MAIN) ;
      DECLARE  (list-of-variables)  FIXED ;
      IF  (sin(x) > 0)
      THEN  statement
      ELSE  statement
END abs ;
```

L'utilisateur peut déplacer le curseur au sein d'une unité manipulable textuellement, comme l'expression "sin(x) > 0", grâce aux commandes "left" et "right". L'édition des expressions s'effectue donc à la manière d'un éditeur de textes "pleine ligne".

La commande "return" permet des déplacements similaires à ceux de "down". Elle permet de positionner le curseur entre les éléments d'une liste, provoquant l'affichage d'un nouvel emplacement vide. Une utilisation ultérieure de "return" sur un emplacement vide d'une liste provoque la destruction de cet emplacement.

Ex :

```
.....
[DECLARE  (list-of-variables) FIXED ;
IF (x>y)
.....
                                        return
.....
DECLARE  (list-of-variables) FIXED ;
[declaration]
IF (x>y)
.....
                                        return
.....
DECLARE (list-of-variables) FIXED ;
[statement]
IF (x>y)
.....
                                        return
.....
DECLARE (list-of-variables) FIXED ;
[IF  (x>y)
.....
```

Certaines unités syntaxiques possèdent des emplacements optionnels, comme l'instruction "do" en PL/1. Lorsque ces emplacements sont vides, ils sont invisibles sur l'écran, et les déplacements de curseur présentés plus haut les ignorent. Seule la commande ".o" (ouvrir) permet le positionnement sur un tel emplacement.

Ex :

```
[DO  i = 1, 10 ;
....
                                        .o
DO  i = 1, 10   WHILE ([condition) ;
....
```

La modification d'un programme nécessite souvent le déplacement d'un groupe d'unités syntaxiques d'un endroit à un autre. La commande "clip" permet de retirer une unité syntaxique d'un emplacement, et la commande "insert" permet sa réinsertion ; "long clip" permet la même opération sur une liste d'unités.

Ex :

```
IF  (x>y)
THEN [x] = 1
ELSE  statement
```

clip

```
IF  (x>y)
THEN [statement]
ELSE  statement
```

return

```
IF  (x>y)
THEN  statement
ELSE [statement]
```

insert

```
IF  (x>y)
THEN  statement
ELSE [x] = 1
```

Les commentaires sont représentés sous forme d'unités, composés de deux emplacements : un pour le texte lui-même, l'autre pour la liste des instructions à commenter. La commande "ellipsis" permet un affichage réduit de ces unités. Ce mécanisme d'élision est différent de celui fourni dans Mentor : pour ce dernier, c'est la profondeur d'imbrication qui détermine l'élision. Pour CPS, c'est l'utilisateur qui choisit la façon d'élider son programme, indépendamment du degré d'imbrication.

Ex :

```
/* calcul du max de x et y */
[IF  (x>y)]
THEN  max = x
ELSE  max = y
```

ellipsis

```
/* calcul du max de x et y */
[.] . .
```

Ce mécanisme permet à l'utilisateur de choisir le niveau de détails qu'il veut pour l'affichage de son programme, ce qui lui permet de voir le plus d'informations pertinentes sur un écran possédant un nombre réduit de lignes. Par ailleurs, cette façon de procéder relève de la programmation par affinements successifs.

L'exécution d'un programme peut être demandée à n'importe quel moment de son écriture ; en particulier, le programme peut être incomplet ou incorrect : l'exécution s'arrête à la rencontre d'une telle erreur. L'utilisateur peut alors modifier le programme, et relancer l'exécution à son point d'arrêt.

L'état d'avancement de l'exécution est visualisé par l'affichage du curseur sur l'instruction en cours. L'utilisateur peut modifier la vitesse de l'exécution, ou choisir le mode "pas à pas". L'utilisation du mécanisme d'élision permet d'éviter la trace des instructions jugées peu intéressantes à observer.

Pendant l'exécution, CPS affiche la valeur des variables du programme que l'utilisateur a sélectionnées. A tout moment, l'utilisateur peut arrêter l'exécution, consulter ou modifier des variables, puis relancer l'exécution (pas nécessairement à l'endroit interrompu).

CPS fournit un exemple remarquable de l'intégration de deux outils d'aide à l'écriture de programmes : édition et interprétation. Ce système est très interactif, et fournit une interface utilisateur de bonne qualité, facile à assimiler et agréable à utiliser. Les commandes se réfèrent à ce que l'utilisateur voit sur l'écran, et non pas à la structure interne du programme.

CPS vérifie la cohérence contextuelle du programme, ce qui le rend beaucoup plus attractif que Mentor, qui se limite à la syntaxe hors-contexte des programmes. Les erreurs sont signalées de façon discrète, et l'utilisateur n'est pas forcé de les corriger immédiatement.

Regrettons cependant que CPS ne fournisse aucune facilité pour écrire de nouvelles commandes et pour rechercher une unité syntaxique particulière, comme le propose Mentor. Ces facilités indispensables pour traiter des gros programmes rendent le système moins apte à une utilisation "professionnelle".

2.4. DISCUSSION.

Nous espérons que l'exposé des grandes fonctions de Mentor et de CPS a permis au lecteur de se faire une idée de l'édition syntaxique et des problèmes qui en découlent.

Deux points de vue opposés ont abouti à la définition de ces éditeurs : pour Mentor, l'utilisateur doit connaître la syntaxe abstraite du langage de programmation utilisé ; les déplacements de l'endroit courant d'édition sont conçus comme un parcours d'arbre. Pour CPS, les déplacements sont définis sur la représentation textuelle du programme ; les quatre commandes principales (up, down, left, right) simulent un dispositif de désignation directe (souris, crayon, etc.) et ressemblent beaucoup aux déplacements proposés par un éditeur de textes orienté écran, ou par un éditeur de formulaires.

Nous pensons que l'édition syntaxique présente de nombreux avantages, mais que le succès de cette approche est essentiellement dépendant de l'interface usager proposée : l'utilisation d'un éditeur syntaxique doit au moins être aussi facile et agréable que celle d'un bon éditeur de textes, sans quoi les programmeurs refusent son emploi. C'est pourquoi nous préférons de beaucoup l'approche proposée par CPS, qui est d'ailleurs celle retenue par les autres éditeurs syntaxiques développés récemment. Une amélioration de cette approche consiste à visualiser l'emplacement courant à l'aide d'un curseur de "surface", mettant en évidence tous les caractères composant l'unité courante. Ceci permet de lever certaines ambiguités, comme par exemple une instruction étiquetée Pascal :

```
50 : x := 1
```

L'utilisateur ne sait pas bien si c'est toute l'instruction ou seulement l'étiquette qui est désignée. Un curseur de surface lève cette ambiguïté :

```
50 :  x := 1          50 : x := 1
instruction           étiquette
```

Le modèle principal de construction dans un éditeur syntaxique paraît agréable à utiliser : l'utilisateur remplit des emplacements par des unités composées d'emplacements, etc. Ce modèle suppose un déplacement automatique du curseur pour être réellement exploitable : un utilisateur construisant un programme ne devrait pas effectuer une seule commande de positionnement pendant cette phase de construction ; ceci permet de simuler une frappe "au kilomètre".

A ce sujet, il faut noter une caractéristique des éditeurs syntaxiques : la différence entre un emplacement simple (la condition d'une instruction conditionnelle) et une liste (les instructions d'une instruction composée). Dans ce dernier cas, il est nécessaire de pouvoir insérer un nombre quelconque d'unités. La solution retenue dans CPS paraît satisfaisante : il y a création implicite d'emplacements vides dans une liste à l'aide de la commande "return", ainsi que lors du remplissage du dernier emplacement d'une unité contenue dans une liste :

```
....
IF (x > y)
THEN  max = x
ELSE [statement ;
x = 1 ;
```

```
                                    max = y
IF  (x > y)
THEN  max = x
ELSE  max = y ;
{statement}
x = 1
```

Une utilisation supplémentaire de la commande "return" supprime facilement l'emplacement inséré.

La notion de liste est définie différemment par CPS et Mentor ou Gandalf : pour ces derniers, une liste est représentée par une unité intermédiaire, manipulable en tant que telle. Pour CPS, cette unité n'existe pas : on passe directement de l'unité englobante au premier élément de la liste par la commande "down". Cette façon de procéder paraît beaucoup plus naturelle et agréable à l'utilisateur. La désignation de la liste complète peut faire l'objet d'une commande spécifique si nécessaire, comme la commande "long clip" de CPS ; la plupart des commandes d'édition ne sont pas freinées par la nécessité de désigner la liste, puis son premier élément.

Cette unité intermédiaire représentant une liste est justifiée par la représentation interne utilisée. Nous verrons au chapitre suivant qu'il est possible de définir une représentation plus adaptée que celle retenue par Mentor et Gandalf.

Un autre aspect fait l'objet de points de vue différents : il s'agit des emplacements optionnels. Un langage de programmation possède fréquemment des unités syntaxiques possédant des parties optionnelles. Il importe de bien distinguer un emplacement optionnel non rempli de l'absence de cet emplacement dans l'unité considérée. Ces deux notions aboutissent en effet à des représentations textuelles distinctes.

Ex : la partie "sinon" d'une instruction conditionnelle Pascal est optionnelle. Il faut distinguer la conditionnelle non remplie :

if condition then instruction else instruction

de la conditionnelle sans partie "sinon" :

if condition then instruction

L'utilisateur doit pouvoir passer simplement d'une forme à l'autre, et pour cela, il est indispensable de ne pas en faire deux unités distinctes dans leur désignation.

Mentor permet toujours le positionnement sur une partie optionnelle, même lorsqu'elle est vide et donc non représentée sur l'écran, ce qui est souvent troublant pour un utilisateur non averti. Pour CPS, les parties optionnelles non remplies ne sont pas visualisées jusqu'à ce que l'utilisateur émette la commande ".o" (ouvrir). C'est cette méthode que nous avons expérimentée dans Adèle : elle présente l'inconvénient que l'utilisateur non averti ne sait jamais sur quelles unités cette commande a un effet. Une autre solution serait peut-être d'étendre la commande "return" aux parties optionnelles : de même qu'elle fait apparaître un emplacement vide entre deux unités d'une liste, de même elle ferait apparaître et disparaître les emplacements optionnels non remplis.

CPS et Mentor permettent des insertions textuelles. Mais CPS effectue une classification des unités : celles qui sont toujours introduites sous forme textuelle et celles qui sont toujours introduites sous forme structurée. Tandis que Mentor permet à l'utilisateur de choisir à tout instant la forme sous laquelle il préfère manipuler une unité. Enfin, pour Gandalf, toute opération s'effectue sous le mode structuré, ce qui est très contraignant pour éditer des expressions. Le compromis le plus satisfaisant consiste peut-être à ne laisser le choix à l'utilisateur que pour les unités simples (expressions) qui ont une représentation textuelle infixée. La construction et l'édition textuelle des expressions paraît la méthode la plus naturelle ; mais il est intéressant, lors de transformations sur tout un programme, de pouvoir considérer une expression comme structurée : les commandes de recherche et de substitution de Mentor sont

de ce point de vue extrêmement puissantes et un bon éditeur syntaxique doit avoir de telles possibilités.

En conclusion à cette discussion, il ressort que l'interface usager d'un éditeur syntaxique est primordiale pour son succès. De ce point de vue, il convient donc de retenir les grands principes de l'éditeur CPS : les commandes se réfèrent à ce que l'utilisateur voit sur son écran, et non pas à la structure interne sous-jacente. Néanmoins, un langage de manipulation de programmes tel que Mentor devient indispensable pour permettre aux utilisateurs de créer leurs propres commandes. Nous pensons donc qu'un bon éditeur syntaxique doit concilier ces deux points de vue.

2.5. L'EDITEUR SYNTAXIQUE DE L'ATELIER ADELE.

Nous ne décrivons que les grandes caractéristiques fonctionnelles de l'éditeur. Le détail des commandes est reporté à l'annexe A1.

Le mode principal de construction d'un programme est l'insertion d'unités syntaxiques, prédéfinies ou créées par l'utilisateur. L'éditeur vérifie à tout instant la validité du programme (analyse contextuelle), et assure son affichage (décompilation) : l'endroit courant d'édition, ainsi que les erreurs contextuelles, sont mis en surbrillance.

Nous appelons "zone de travail" un espace contenant un fragment de programme Pascal. La zone principale est celle contenant le programme en cours d'édition. L'utilisateur peut créer des zones annexes qu'il peut remplir par des fragments de programme, réutilisables en tant qu'unités syntaxiques. L'analyse contextuelle n'est effectuée que sur la zone principale.

Cette notion de zone de travail est très importante pour l'utilisateur, car elle apporte une grande souplesse à la manipulation des unités syntaxiques.

L'affichage sur l'écran correspond toujours à la zone de travail courante. Il est possible de changer de zone de travail par demande de "positionnement" sur une nouvelle zone.

A l'intérieur d'une zone de travail, nous définissons trois types de positionnement possibles :

- le positionnement direct par désignation d'un numéro de ligne (affiché sur l'écran),
- le positionnement relatif par déplacement structurel (exemple : monter, descendre, suivant, précédent),
- le positionnement par recherche d'une occurrence de zone de travail, prédéfinie ou non.

L'utilisateur a donc toujours la notion d'endroit courant dans une zone de travail ; cet endroit courant peut être modifié par une demande de positionnement sur une autre unité syntaxique, et sert d'argument à certaines commandes d'édition.

La manipulation des unités syntaxiques ainsi désignées est effectuée par des primitives (opérateurs) telles que :

- détruire l'unité syntaxique courante,
- déplacer un groupe d'unités syntaxiques à l'endroit courant,
- copier un groupe d'unités syntaxiques à l'endroit courant.

Une unité ou un groupe d'unités syntaxiques que l'on désire copier ou déplacer ne constitue pas toujours la totalité d'une zone de travail, mais peut en être un sous-ensemble ; il semble donc utile de donner à l'utilisateur la possibilité de repérer ces unités syntaxiques de façon à pouvoir les désigner ultérieurement sans avoir à se positionner dessus. Nous donnons le nom de "marque" à ce repère. Comme l'endroit courant, la marque peut servir d'argument implicite à certaines commandes.

La principale opération de construction d'unités syntaxiques étant la copie de modèles prédéfinis, il est naturel qu'après une copie, le positionnement se fasse automatiquement (aussi souvent que possible) sur le prochain emplacement "vide". Ceci évite à l'utilisateur de demander explicitement le positionnement sur le prochain emplacement à remplir.

L'opération de copie s'effectue nécessairement dans un emplacement vide. Dans une liste, l'éditeur fournit donc une commande d'insertion d'un emplacement vide. Signalons que la frappe de l'opérateur de copie est optionnelle : l'utilisateur a seulement besoin de donner le nom de l'objet qu'il veut copier.

Exemple :

action utilisateur	Etat visuel de l'information (la partie encadrée désigne l'endroit courant)
	(1) [if] (* condition *) then (2) (* instruction *)
"positionnement sur la ligne numéro 2"	(1) if (* condition *) then (2) (* [instruction] *)
"copie de l'instruction prédéfinie begin-end"	(1) if (* condition *) then (2) begin (3) (* [instruction] *) (4) end (l'endroit courant devient automatiquement la première instruction à remplir et l'utilisateur peut directement copier une autre instruction).
"copie du modèle prédéfini while"	(1) if (* condition *) then (2) begin (3) while (* [condition] *) do (4) (* instruction *) ; (5) (* instruction *) (6) end (On remarquera que la copie d'une instruction "while" dans l'emplacement (* instruction *) de la ligne 3, crée une nouvelle instruction, sur laquelle on sera positionné automatiquement dès que sera terminé le remplissage de la condition et de l'instruction de l'itération).

Ces opérations de copie de modèles prédéfinis peuvent paraître fastidieuses à l'utilisateur, lors de la construction des expressions. En effet, la construction de l'expression : (a+1)-(b*4) nécessite les copies successives des modèles prédéfinis : "-", "+", "*", c'est-à-dire que l'expression est rentrée sous un mode préfixé, peu conforme à l'usage.

D'où la notion de mode "introduction de texte" qui permet à l'utilisateur de taper directement du texte source Pascal, qui est analysé avant d'être inséré à l'endroit courant.

L'utilisateur a la possibilité d'introduire du texte à tout niveau, sur une expression comme sur le programme entier. Mais le texte introduit est actuellement perdu en cas d'erreur syntaxique, ce qui limite en pratique l'usage de ce mode aux unités simples comme les expressions. Enfin, il n'est pas possible actuellement d'éditer textuellement une unité déjà construite.

L'éditeur syntaxique offre également un système d'aide en permettant à l'utilisateur d'afficher à tout moment :

- l'endroit courant (repéré sur l'écran par un symbole spécial),
- la liste des unités syntaxiques prédéfinies,
- la liste des noms de zone de travail définies par l'utilisateur,
- le contenu d'une zone de travail,
- la liste des opérateurs disponibles sous l'éditeur,
- le mode d'utilisation d'un opérateur.

Ce système d'aide est complété par la possibilité d'annuler l'action précédemment effectuée et de se retrouver dans la situation qui était celle avant le traitement de l'action annulée. L'utilisateur se voit ainsi accorder une sorte de "droit à l'erreur" et peut revenir sur une décision (le résultat d'une action étant immédiatement visible sur l'écran).

- L'annulation d'une commande de positionnement permet de retrouver directement le précédent endroit courant sans obliger l'utilisateur à se rappeler où il était positionné, ni à décrire le moyen de s'y repositionner.

- L'annulation d'une demande de copie ou de déplacement d'une unité syntaxique peut être considérée comme la commande "inverse", mais est d'usage beaucoup plus souple pour l'utilisateur qui n'a alors pas besoin de préciser les paramètres de cette commande "inverse".

Ex : annuler la commande "copier" est équivalent à la commande "détruire", annuler la commande "déplacer" une unité syntaxique localisée en "adrl" à l'adresse "adr2" est équivalent à "déplacer" l'unité localisée en "adr2" à l'adresse "adrl".

- L'annulation de la commande "détruire" représente certainement la possibilité la plus intéressante car elle permet de remettre en place une unité syntaxique qui est censée être disparue et l'utilisateur n'a pas besoin de la recréer.

Exemple :

Action Utilisateur	Etat visuel de l'information
	(1) while i > 0 do (2) begin (3) p(i) ; (4) i := i - 1 (5) end
"détruire"	(1) while i > 0 do (2) begin (3) p(i) ; (4) (*instruction*) (5) end
"annuler"	(1) while i > 0 do (2) begin (3) p(i) ; (4) i := i - 1 (5) end

Notons toutefois que l'action d'annulation concerne obligatoirement la dernière action ayant entraîné des modifications de la représentation interne et que certaines actions, telles que la sauvegarde du programme dans un fichier, ne supportent pas cette possibilité.

Voyons maintenant comment un utilisateur peut exprimer les actions qu'il veut entreprendre. Traditionnellement, une action utilisateur (commande) consiste à effectuer une opération sur un objet et est décrite par la frappe d'une commande, désignant un opérateur (exemple : opérateur de substitution), suivie ou non d'arguments, désignant les objets sur lesquels doit porter l'action ; c'est l'approche "opérateur-objet". Pour l'éditeur, afin de respecter les spécifications externes du médiateur, nous partons du principe inverse : entreprendre une action, c'est d'abord désigner un objet, puis un opérateur acceptable pour cet objet ; c'est l'approche "objet-opérateur".

Souvent, l'action entreprise ne porte que sur un objet, et cet objet est désigné de façon implicite. En effet, la désignation d'un objet peut résulter d'un positionnement antérieur sur cet objet. Ainsi, l'opérateur "détruire" concernera souvent l'unité syntaxique désignée par l'endroit courant.

Mais certains opérateurs nécessitent la désignation de deux objets. Dans ce cas, le second objet pourra être désigné explicitement après la frappe de l'opérateur. Ainsi, l'opérateur "déplacer" nécessite deux renseignements : l'unité syntaxique à déplacer et la destination. La destination est alors implicitement désignée par l'endroit courant et l'unité syntaxique à déplacer peut être indiquée ou prise par défaut (ce sera alors la marque).

En résumé, l'éditeur propose des commandes de construction et de modification structurées de programmes Pascal. La maquette réalisée possède de nombreuses limitations. En particulier, l'aspect textuel d'un programme n'est pas suffisamment pris en compte, particulièrement dans les commandes de positionnement et de modification. Mais le travail qui reste à effectuer est un problème de réalisation plutôt que de conception.

CHAPITRE III

REPRESENTATIONS INTERNES DE PROGRAMMES ET ARBRES ABSTRAITS

Ce chapitre développe ce qu'est une représentation interne de programmes dans un atelier de génie logiciel. Après avoir étudié les qualités requises par une RI, nous montrons que la notion d'arbre abstrait est bien adaptée à notre problème. Nous évoquons la théorie des ramifications pour aboutir à une notion d'arbre légèrement différente de celle habituellement rencontrée dans les éditeurs syntaxiques, et notamment Mentor. Pour cela nous introduisons la notion de rameau, permettant d'exprimer plus succinctement une liste d'unités syntaxiques. Nous terminons cette définition d'arbre abstrait en évoquant quelques problèmes rencontrés par les concepteurs d'une syntaxe abstraite.

La deuxième partie de ce chapitre est consacrée aux opérations qu'il est possible d'effectuer sur une RI. Nous essayons de montrer comment décrire un outil utilisant une RI. Nous illustrons enfin nos propos en détaillant la RI de l'atelier Adèle.

Tout au long de ce chapitre, nous nous efforçons de trouver des solutions générales au problème de représentation d'un programme : même si nos exemples sont le plus souvent tirés du langage Pascal, les solutions peuvent s'appliquer à tout langage de programmation. De plus, nous pensons moins à un atelier pour un langage donné qu'à un générateur d'ateliers, paramétré par le langage. Cette idée sert de fil directeur à tout le chapitre.

La réalisation effectuée dans l'atelier Adèle est loin d'être aussi générale que nous l'aurions souhaité. Mais c'est cela qui nous a permis de mieux comprendre les problèmes que nous évoquons.

3.1. CRITERES DE CHOIX D'UNE RI.

Il paraît a priori difficile de concevoir une RI satisfaisant les besoins des outils, existants ou à venir, d'un atelier : chaque outil a une vision particulière d'un programme et ne traite qu'une partie des informations qu'il contient. Il s'adapte donc tant bien que mal au modèle proposé, en ne retenant que les informations qui l'intéressent.

Il est donc intéressant de décomposer une RI en un squelette connu de tous, auquel sont rattachées des décorations spécifiques d'un outil ou d'un groupe d'outils. Nous pensons que ce squelette doit posséder les qualités suivantes :

(a) Equivalence sémantique.

Tout programme valide pour le langage considéré peut être représenté. Réciproquement, le squelette est suffisant pour reconstruire un texte source, à la mise en page près (décompilation). La décompilation est indispensable d'une part pour l'interaction homme-machine, et d'autre part pour alimenter des outils extérieurs à l'atelier. Les décorations peuvent mémoriser les commentaires, les directives de mise en page, et permettent ainsi, si nécessaire, une reconstruction fidèle du texte source.

(b) Validité syntaxique.

Le squelette peut représenter un programme syntaxiquement incomplet ou incorrect (par syntaxe, nous entendons syntaxe hors-contexte et contextuelle). Ceci est rendu nécessaire par le processus de construction d'un programme. Il est préférable d'assurer la cohérence de la syntaxe hors contexte, comme le permettent les éditeurs syntaxiques ; par contre, imposer à tout instant la validité contextuelle du programme serait inacceptable pour les utilisateurs de l'atelier (nous avons vu au chapitre 2, que l'éditeur CPS signale les erreurs contextuelles, mais ne force pas leur correction). Et l'atelier peut avoir un éditeur "simplifié", comme Mentor, qui n'effectue aucune vérification contextuelle.

(c) Extraction.

Il est possible d'isoler du squelette des fragments de programme constituant une unité syntaxique du langage, comme une déclaration, une instruction, etc. Ceci permet des opérations diverses, dont la modification

structurée du programme ; le squelette doit être un reflet naturel des entités manipulées par l'éditeur syntaxique.

(d) Définition.

Le squelette peut être décrit dans un formalisme compréhensible par l'homme et adapté à divers traitements automatiques, paramétrés par la syntaxe du langage de programmation. Deux catégories d'utilisateurs sont concernées par ce formalisme : l'utilisateur et le concepteur d'un outil ; tous deux ont besoin d'une description non ambiguë de la RI.

(e) Implémentation.

La RI est considérée comme une structure abstraite, manipulée indépendamment de sa nature réelle dans la machine. Elle permet cependant des implémentations efficaces, et indépendantes du langage de programmation qu'elle représente. Cette dernière contrainte assure une ouverture ultérieure de l'atelier à d'autres langages.

(f) Archivage.

La RI d'un programme est archivée sous une forme indépendante de la machine support de l'atelier. Ceci permet d'une part l'évolution de l'implémentation, et d'autre part le transfert de programmes vers d'autres environnements.

3.2. QUELQUES CHOIX POSSIBLES.

L'idée d'utiliser des représentations "intermédiaires" des programmes n'est pas récente, et est apparue avec la notion de compilation : plutôt que de considérer un compilateur comme un bloc monolithique, admettant un programme source et produisant un code objet, il est naturel de décomposer (au moins conceptuellement) ce traducteur en plusieurs modules formant une chaîne de production. Les entrées et sorties de ces modules sont des représentations intermédiaires du programme source, dont l'existence globale peut n'être que conceptuelle, si elles sont produites au fur et à mesure des besoins. Cette décomposition d'un compilateur facilite sa compréhension, sa spécification et sa mise au point, et rend possible la réutilisation de certains composants sur des machines différentes.

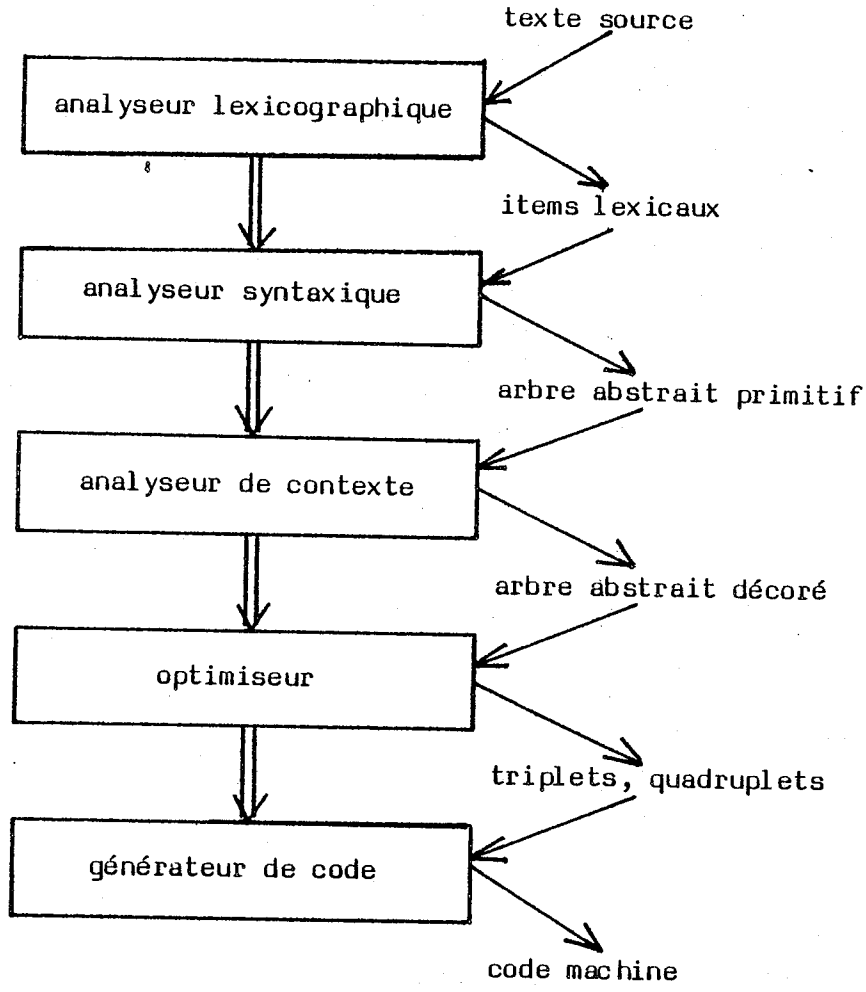


Fig. 3.1. Organisation possible d'un compilateur.

L'examen des qualités requises nous a conduits à rejeter des RI de "bas niveau", proches des langages machines : elles ne reflètent plus la sémantique du programme, mais sa traduction dans un langage particulier, dans lequel interviennent des notions opératoires particulières à un dispositif physique. De plus, la pauvreté sémantique des langages machines rend la traduction non bijective.

D'un autre côté, le texte source n'apparaît pas non plus comme une représentation facilement manipulable : la plupart des outils devraient commencer par une phase d'analyse lexicale et syntaxique avant de pouvoir manipuler une entité du langage. Ils auraient alors leur propre représentation des programmes, ce qui serait contraire au principe d'unicité.

Le compromis le plus intéressant se situe donc après la phase d'analyse, et avant la phase de traduction dans des langages moins riches.

L'analyse syntaxique produit naturellement un arbre : l'arbre de la syntaxe concrète d'un langage (arbre syntaxique). Les noeuds de l'arbre sont étiquetés par les symboles non terminaux de la grammaire hors-contexte du langage, tandis que les feuilles en représentent les symboles terminaux (unités lexicographiques). Les non-terminaux sont introduits pour permettre une analyse syntaxique déterministe du programme source. On retrouve donc dans l'arbre syntaxique des informations de présentation textuelle du programme, qui ne reflètent pas sa sémantique mais la technique d'analyse utilisée, descendante ou ascendante.

Exemple : considérons un langage traitant des expressions arithmétiques, en tenant compte de la priorité habituelle des opérateurs. Nous pouvons définir deux grammaires de ce langage, permettant respectivement une analyse descendante -LL(1)- et ascendante -LR(1)-. Nous présentons les arbres syntaxiques respectifs, correspondant au texte source : "cte * id".

E ::= T X
X ::= + T X
X ::= ε
T ::= P Y
Y ::= * P Y
Y ::= ε
P ::= cte
P ::= id
P ::= (E)

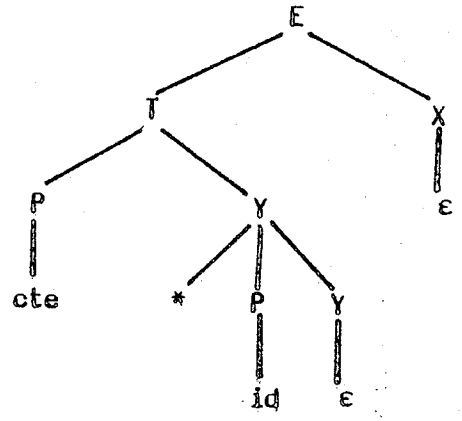


Fig. 3.2. Grammaire LL(1).

E ::= E + T
E ::= T
T ::= T * P
T ::= P
P ::= cte
P ::= id
P ::= (E)

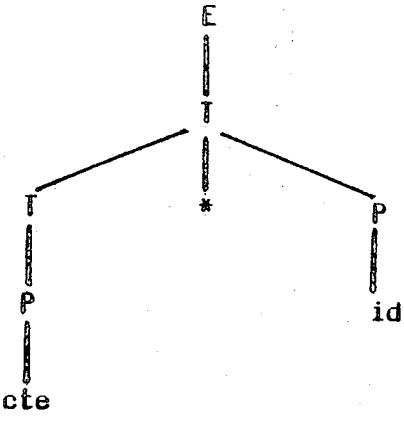


Fig. 3.3. Grammaire LR(1).

Lorsqu'on s'intéresse essentiellement à la sémantique d'un programme, il est naturel de retenir l'arbre de la syntaxe abstraite (arbre abstrait) d'un programme comme représentation interne. Le terme "abstrait" indique le souci d'éliminer les détails de représentation externe du programme. Grossièrement, l'arbre abstrait peut se déduire de l'arbre concret :

- en élaguant les noeuds intermédiaires inutiles et les feuilles correspondant à du "sucre syntaxique" (par exemple, les mots-clés et les signes de ponctuation),

- en étiquetant les noeuds restants par les noms significatifs des feuilles élaguées.

Exemple : le texte "cte * id" est réprésenté par l'arbre abstrait :

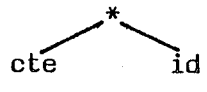


Fig. 3.4. Arbre abstrait.

Cet exemple montre bien la concision de l'arbre abstrait par rapport à l'arbre syntaxique, sans perte d'information utile : l'arbre syntaxique pourrait, si nécessaire, être reconstitué à partir de l'arbre abstrait, ne serait-ce qu'en reconstruisant le texte source...

3.3. RI FONDEES SUR LES ARBRES ABSTRAITS.

3.3.1. Introduction.

Dans le chapitre 3.2, nous avons défini de façon intuitive la notion d'arbre abstrait. Cette notion a été utilisée avec succès dans la plupart des systèmes que nous avons évoqués au chapitre 2 (le lecteur intéressé pourra notamment se reporter aux références des systèmes Mentor et Gandalf) : un arbre abstrait est le reflet naturel des entités externes que manipule l'utilisateur d'un éditeur syntaxique.

Les noeuds de l'arbre correspondent aux unités syntaxiques, et les branches aux emplacements pouvant contenir une unité. Les commandes de positionnement de Mentor correspondent à un positionnement dans l'arbre ; celles de CPS effectuent un parcours préfixé de l'arbre. Un éditeur syntaxique peut donc être défini conceptuellement comme un éditeur d'arbres abstraits, même s'il utilise une autre représentation des programmes.

C'est pourquoi il est nécessaire de définir précisément ce qu'est un arbre abstrait : nous avons utilisé le formalisme des ramifications élaboré par Pair et Quéré <Pair 68> pour donner une définition algébrique de l'univers des arbres abstraits.

Les éditeurs syntaxiques actuels sont encore loin d'être parfaits. Notamment, leur comportement vis-à-vis des listes d'unités syntaxiques nous paraît inadapté. Considérons par exemple l'instruction répétitive Pascal et l'arbre abstrait associé :

```
repeat
    x := 2 * x ;
    y := y - 1
until  x > y
```

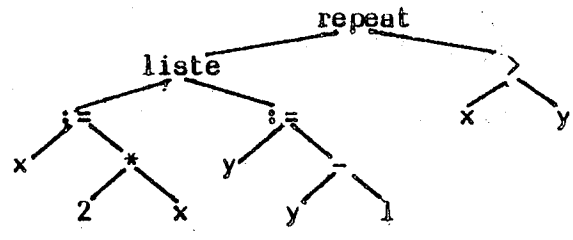


Fig. 3.5. Première définition d'arbre abstrait.

Le noeud "liste" possède un nombre quelconque d'arbres fils représentant des instructions ; mais il n'a aucun équivalent dans la représentation externe du programme, et cependant l'utilisateur des éditeurs syntaxiques tels que Mentor et Gandalf doit connaître son existence. Lorsque le curseur d'édition

est positionné sur ce noeud, c'est l'ensemble des instructions filles qui est désigné.

Nous pensons donc que la notion de noeud "liste" ne doit pas exister dans la définition d'arbre abstrait (c'est d'ailleurs ainsi que peut être décrit l'éditeur CPS). Dans notre exemple, nous préférons définir l'instruction répétitive comme possédant deux branches, dont la première peut posséder un nombre quelconque d'instructions : nous appelons rameau une telle branche.

```
repeat
     x := 2 * x ;
     y := y - 1
until  x > y
```

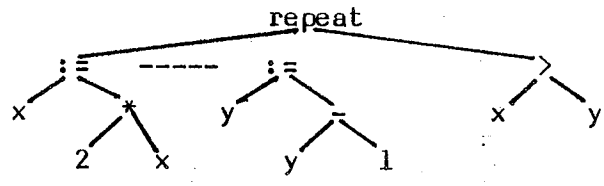


Fig. 3.6. Deuxième définition d'arbre abstrait.

Pour laisser au lecteur la possibilité de juger ce choix, nous avons détaillé ces deux définitions d'arbre abstrait. La suite de cette thèse utilisera la notion de rameau, mais aurait pu être écrite sans cette notion.

3.3.2. Algèbre des ramifications et bigrammaires régulières.

La définition classique d'un arbre le présente comme un graphe connexe sans cycle. Pair et Quéré ont introduit le formalisme des ramifications pour définir algébriquement la notion de structure arborescente. Ceci permet de définir des grammaires de ramifications engendrant des bilangages, selon une démarche analogue à celle utilisée pour définir les notions usuelles de monoïde (ensemble des mots sur un alphabet), de grammaire et de langage.

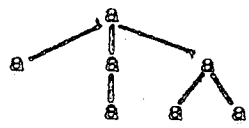
Nous utiliserons des définitions plus intuitives que rigoureuses, afin d'en faciliter la compréhension. Parmi la classe des bigrammaires, nous ne nous intéressons qu'aux bigrammaires régulières. Le lecteur intéressé par plus de détails pourra se reporter à <Pair 68> et aux thèses de <Berlioux 72>, <Marchand 74> et <Simonet 81>.

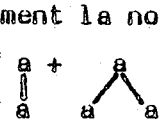
Soit V un ensemble. On appelle <u>V-binoïde</u> tout ensemble possédant les deux lois suivantes :

- Concaténation (notée +) : loi interne associative permettant de juxtaposer deux arbres. Cette loi possède un élément neutre, appelé arbre vide, et noté Λ.

- Enracinement (noté x) : loi externe à opérateurs dans V.

<u>Ex</u> : pour $V = \{a\}$, on peut définir par "a x (a + axa + ax(a + a))" l'arbre :



La notation visuelle  correspond à a x a. Nous utiliserons également la notation semi-visuelle : a x (a + ).

Deux V-binoïdes B et B' sur le même ensemble V sont dits <u>homomorphes</u> s'il existe une application f de B dans B', telle que :

$$f(x + y) = f(x) + f(y)$$
$$f(a \times x) = a \times f(x).$$

On montre qu'il existe un V-binoïde, noté $\hat{V}$, défini à un isomorphisme près par les conditions :

- tout élément r de $\hat{V}$, $r \neq \Lambda$, s'écrit de manière unique :
  $r = (a \times r') + r''$ avec $a \in V$, $r' \in \hat{V}$, $r'' \in \hat{V}$.
- il existe une fonction $\nu : \hat{V} \longrightarrow N$ telle que :
  $\nu(\Lambda) = 0$
  $\nu(a \times r) > \nu(r)$
  $r \neq \Lambda \Longrightarrow \nu(r + s) > \nu(s)$
  $s \neq \Lambda \Longrightarrow \nu(r + s) > \nu(r)$

On appelle <u>ramification</u> un élément de $\hat{V}$ et <u>bilangage</u> un sous-ensemble de $\hat{V}$. Un <u>arbre</u> est une ramification ayant au plus une racine.

Le mode principal de raisonnement dans $\hat{V}$ est donné par le schéma de récurrence : $P(\Lambda)$ et $(\forall a, \forall r, \forall s, P(r)$ et $P(s) \Longrightarrow P(a \times r + s)) \Longrightarrow (\forall r, P(r))$. Ceci permet de construire quelques fonctions usuelles dont nous donnons quelques exemples (V^* désigne le monoïde sur V) :

- Mot des racines $\rho : \widehat{V} \longrightarrow V^*$: $\rho(\Lambda) = \Lambda$, $\rho(a \times r + s) = a\,\rho(s)$
- Mot des feuilles $\phi : \widehat{V} \longrightarrow V^*$: $\phi(\Lambda) = \Lambda$, $\phi(a \times r + s) =$ (si $r = \Lambda$ alors $a\phi(s)$ sinon $\phi(r)\phi(s)$)
- Nombre de noeuds $n : \widehat{V} \longrightarrow N$: $n(\Lambda) = 0$, $n(a \times r + s) = 1 + n(r) + n(s)$
- Hauteur $h : \widehat{V} \longrightarrow N$: $h(\Lambda) = 0$, $h(a \times r + s) = \sup(1+h(r), h(s))$

Soient V et T deux ensembles disjoints. On appelle <u>V-binoïde de base T</u>, et l'on note $\widehat{V(T)}$, le bilangage sur $\widehat{V \cup T}$ dont les ramifications n'ont d'occurrences des éléments de T qu'aux feuilles. Cette notion est utile pour définir les bigrammaires régulières.

On appelle grammaire de ramifications régulières, ou <u>bigrammaire régulière</u>, tout quadruplet (Vt, Vn, S, P), tel que :

- Vt et Vn sont des ensembles finis (vocabulaires terminal et non terminal)
- S est un élément de Vn, l'axiome
- P est un ensemble fini de règles de production de la forme :
  $A \longrightarrow r + B$ ou $A \longrightarrow r$, avec $A, B \in Vn$ et $r \in Vt(Vn)$
  (les éléments de V n'ont d'occurrences qu'aux feuilles de r).

Une <u>dérivation</u> p ==> q se définit en remplaçant un non-terminal par une partie droite de règle. La fermeture réflexive et transitive de ==> se note ==>*.

Le bilangage L(G), <u>engendré</u> par une bigrammaire régulière G, est :
$L(G) = \{r \in \widehat{Vt} \;/\; S \Longrightarrow^* r\}$.

Un bilangage est <u>régulier</u> s'il peut être engendré par une bigrammaire régulière.

<u>Exl</u> : Vt = {a, b}
Vn = {S, T}
P = {S --> Λ, S --> b + S, S --> a + S, T --> b + S, T --> a + S}
(a + S : T ; a + S : T)

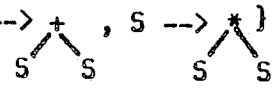
Le bilangage engendré est l'ensemble des arbres sur a, b n'ayant que b comme feuilles :

L(G) = { Λ, b, a(b) , a(b b) , a(a(b b) b) , ...}

Ex2 : (bilangage des expressions arithmétiques du chapitre 3.2).

Vt = {+, *, id, ct}

Vn = {S}

P = {S --> id, S --> ct, S -->  , S --> }

3.3.3. Première définition de grammaire abstraite.

A priori, tout bilangage régulier représente la syntaxe abstraite d'un certain langage, et ses éléments sont donc des arbres abstraits. Cependant, il est souvent plus simple de présenter un langage comme composé d'entités fonctionnelles : une instruction d'affectation est composée d'une variable et d'une valeur arithmétique, une instruction composée est la concaténation d'un nombre quelconque d'instructions simples. La description d'une syntaxe abstraite est donc plus claire si elle met ce concept en évidence.

Intuitivement, l'affectation est une opération binaire, la conditionnelle est ternaire, et l'instruction composée est d'arité variable. Cette notion d'arité, fixe ou variable, n'apparaît pas dans une bigrammaire régulière ; nous allons donc définir un formalisme engendrant des bilangages réguliers, et exprimant la notion d'arité. Ce formalisme est équivalent à ceux proposés dans IDL <Goos 81>, <Nestor 81>, Mentor et Gandalf. Nous employons la terminologie de phylum et d'opérateur, terminologie empruntée au système Mentor.

Une grammaire abstraite est un quadruplet (Vt, Vn, S, P) tel que :

- Vt est le vocabulaire terminal (les opérateurs)
- Vn est le vocabulaire non terminal (les phylums)
- S est un élément de Vn, l'axiome
- P est un ensemble de règles de production, de la forme :

(a)	X ::= y	$X \in Vn$, $Y \in Vn$
(b)	X ::= t	$X \in Vn$, $t \in Vt$
(c)	t --> X1, X2, ..., Xn	$t \in Vt$, $Xi \in Vn$ $(n > 0)$
(d)	t --> X*	$t \in Vt$, $X \in Vn$
(e)	t --> X+	$t \in Vt$, $X \in Vn$

On impose de plus que chaque opérateur soit partie gauche d'une et d'une seule règle. Cette contrainte fixe la nature de l'arité (binaire, ternaire, ..., variable) d'un opérateur.

Les règles (a) et (b) expriment le fait qu'un phylum est un sous-ensemble (non vide) d'opérateurs. L'écriture de ces règles peut être condensée en pratique :

X ::= X1, X ::= X2, ..., X ::= Xn peut s'écrire : X ::= X1, X2, ..., Xn..

La règle (c) désigne un opérateur n-aire. Chacun de ses opérandes (ses fils), doit être un opérateur appartenant à un domaine précis, déterminé par le phylum de même rang.

Les règles (d) et (e) désignent un opérateur d'arité variable. Il pourra avoir un nombre quelconque de fils (éventuellement aucun pour la règle (d), au moins un pour la règle (e)), appartenant tous au même phylum.

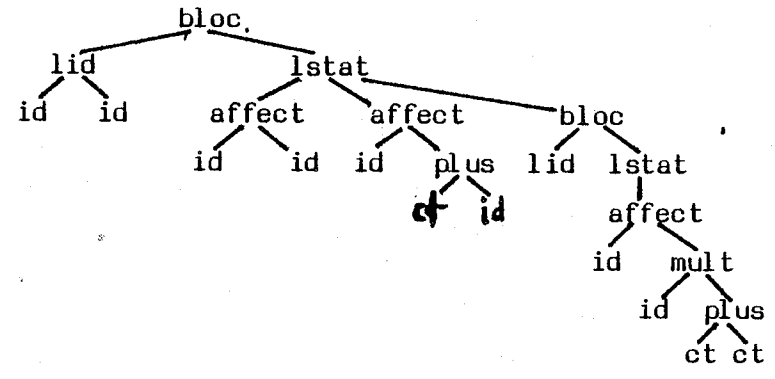
Un abus de notation nous autorise à assimiler un opérateur x au phylum {x} dans les règles (c), (d), (e).

Ex : Syntaxe abstraite d'un langage à structure de blocs.

Vt = {id, ct, plus, mult, affect, bloc, lid, lstat}
Vn = {PROG, STAT, EXP} (axiome : PROG)
PROG ::= bloc
STAT ::= bloc, affect
EXP ::= id, ct, plus, mult
bloc --> lid, lstat
lid --> id*
lstat --> STAT*
affect --> id, EXP
plus --> EXP, EXP
mult --> EXP, EXP
id -->
ct -->

Un exemple d'arbre engendré est le suivant (pour plus de clarté, nous indiquons également le programme concret) :

```
begin
    id, id ;
    id := id ;
    id := ct+id ;
    begin
        id := id * (ct + ct) ;
    end ;
end.
```



Le langage engendré par une grammaire abstraite est un bilangage régulier. Ceci se démontre en fournissant un procédé de transformation de grammaire abstraite en bigrammaire régulière. Ce procédé ne modifie pas le langage engendré :

- Remplacer les parties droites des règles $X ::= Y$ ($Y \in Vn$) par l'expansion de Y. A la fin de cette étape, il ne reste plus de règles de type (a) ; tous les phylums sont définis par des règles de la forme : $X ::= t$ ($X \in Vn$, $t \in Vt$).

- Remplacer les règles $t \longrightarrow X^*$ par les règles (où X' est un nouveau non-terminal) :

  $X' \longrightarrow \Lambda$

  $X' ::= u + X'$ pour tout $u \in Vt$ vérifiant $X ::= u$

  $t \longrightarrow X'$

- Remplacer les règles $t \longrightarrow X+$ par les règles (où X' est un nouveau non-terminal) :

  $X' \longrightarrow u$ pour tout $u \in Vt$ vérifiant $X ::= u$

  $X' \longrightarrow u + X'$

  $t \longrightarrow X'$

- Remplacer les règles :

  $X ::= t$ et $t \longrightarrow X1, X2, \ldots, Xn$ ($t \in Vt$, $Xi \in Vn$)

  par $X \longrightarrow t \times (X1 + X2 + \ldots + Xn)$

- Remplacer les règles temporaires :

  $X ::= t + Y$ et $t \longrightarrow Z$ ($t \in Vt$, $X, Y, Z \in Vn$)

  par $X \longrightarrow t \times Z + Y$ et $t \longrightarrow Z$

- Supprimer les règles : $t \longrightarrow X1, X2, \ldots, Xn$ ($t \in Vt$).

La grammaire abstraite de l'exemple précédent est transformée en bigrammaire régulière :

Vt = {id, ct, plus, mult, affect, bloc, lid, lstat}

Vn = {PROG, STAT, EXP, LID, LSTAT, ID, ID', STAT'} (axiome : PROG)

PROG --> bloc
(LID, LSTAT)

STAT --> bloc
(LID, LSTAT)

STAT --> affect
(ID, EXP)

EXP --> id

EXP --> ct

EXP --> plus
(EXP, EXP)

EXP --> mult
(EXP, EXP)

ID --> id

LID --> lid
ID'

ID' --> Λ

ID' --> id + ID'

LSTAT --> lstat
STAT'

STAT' --> Λ

STAT' --> bloc + STAT'
(LID, LSTAT)

STAT' --> affect + STAT'
(ID, EXP)

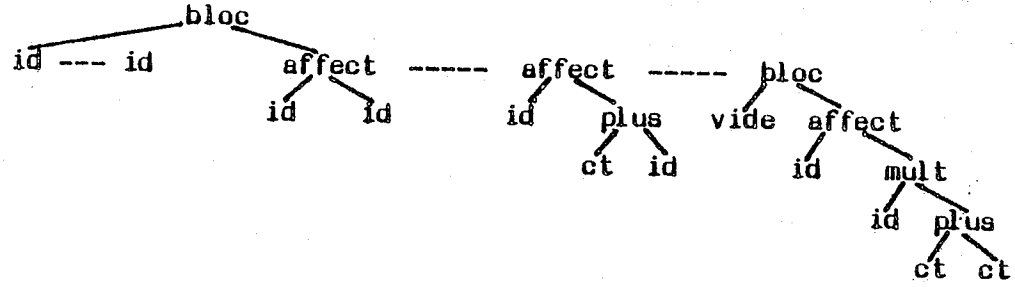
Le langage engendré par une grammaire abstraite est donc un bilangage régulier. Réciproquement, il existe des bigrammaires régulières ne pouvant pas s'écrire sous forme de grammaires abstraites : ces grammaires ne fixent pas l'arité d'un opérateur.

Ex : $G = (\{a\}, \{S\}, S, \{S \to a, S \to a_a\})$.

Remarque : on pourrait sans doute montrer l'équivalence en utilisant un formalisme étendu de grammaire abstraite, permettant de fixer les nombres minimaux et maximaux de fils d'un opérateur d'arité variable.

3.3.4. Deuxième définition de grammaire abstraite.

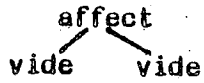
Examinons l'exemple de grammaire abstraite développé au paragraphe précédent : l'obligation de rendre l'arité d'un opérateur, soit fixe, soit variable, nous a conduits à définir deux opérateurs de liste : lid et lstat, que nous voudrions supprimer d'après les raisons évoquées au paragraphe introductif 3.3.1. Nous voulons donc présenter l'arbre abstrait de l'exemple sans ces opérateurs :



Pour cela, nous introduisons une nouvelle loi associative, notée ---, permettant de composer des arbres. Nous appelons rameau le résultat de cette loi.

Dans le formalisme des ramifications, arbre et ramification étaient déjà deux notions distinctes. Avec cette nouvelle loi, nous obtenons trois notions distinctes : arbre, rameau et ramification. Une ramification est la concaténation de rameaux ou l'arbre . Un arbre reste l'enracinement d'un opérateur et d'une ramification.

Nous avons de plus introduit un nouvel opérateur : l'opérateur "vide". Il permet de représenter des rameaux vides, c'est-à-dire des arbres incomplets. Par exemple, une affectation incomplète est représentée par le rameau :



Dans un rameau, on pourra trouver un nombre quelconque d'arbres réduits à l'opérateur vide. Un éditeur syntaxique pourra ainsi mémoriser, à la demande de l'utilisateur, un nombre quelconque d'emplacements à remplir dans une liste.

Cet opérateur vide permet également de représenter la notion d'emplacement optionnel, comme la partie sinon de l'instruction conditionnelle Pascal :

Nous n'établissons pas de différence de représentation entre un rameau optionnel non rempli et un rameau incomplet. Nous verrons plus loin que si un outil désire effectuer cette distinction, il lui suffira d'annoter l'opérateur vide (exemple : décompilation de la partie sinon ; suivant le cas il faut ou il ne faut pas afficher le mot-clé "else").

L'opérateur vide paraît donc indispensable pour l'utilisation des arbres abstraits dans un atelier. Aussi, par convention, il est inutile de le définir explicitement dans une grammaire abstraite : tout phylum le contiendra implicitement.

Une grammaire abstraite est un quadruplet (Vt, Vn, S, P) tel que :

- Vt est le vocabulaire terminal (les opérateurs)
- Vn est le vocabulaire non terminal (les phylums)
- S est un élément de Vn, l'axiome
- P est un ensemble de règles de la forme :

(a) X ::= Y ou X ::= t — X, Y ∈ Vn, t ∈ Vt

(b) t --> U1, U2, ..., Un — (n > 0) où les Ui sont de la forme :

X	X ∈ Vn	(rameau simple)
<X>	X ∈ Vn	(rameau optionnel)
X*	X ∈ Vn	(rameau composé de p éléments, p > 0)
X+	X ∈ Vn	(rameau composé de p éléments, p > 1)

De plus, chaque opérateur est partie gauche d'une et d'une seule règle.

Nous pourrons utiliser les mêmes abus de notation que dans le paragraphe précédent.

Ex : grammaire du langage à structure de blocs :

Vt = {id, ct, plus, mult, affect, bloc}
Vn = {PROG, STAT, EXP}
PROG ::= bloc
STAT ::= bloc, affect
EXP ::= id, ct, plus, mult
bloc --> id*, STAT*
affect--> id, EXP
plus --> EXP, EXP
mult --> EXP, EXP
id -->
ct -->

Cette grammaire produit bien l'arbre du début du paragraphe.

Il est facile de revenir au formalisme précédent, et donc aux bigrammaires régulières :

- pour chaque phylum X, rajouter la règle X ::= vide
- pour chaque règle t --> U1, U2, ..., Un :
  - si Ui = <X>, remplacer Ui par X
  - si Ui = X* ou X+, introduire un nouveau terminal u', remplacer Ui par u', et ajouter la règle u'--> X* ou u' --> X+
- ajouter "vide" au vocabulaire terminal.

3.3.5. Arbres abstraits et annotations.

Un programme peut théoriquement être représenté par un arbre abstrait. Examinons cependant les notions d'identificateur et de constante : il est possible de définir un identificateur comme une suite d'opérateurs représentant un caractère alphanumérique :

```
id       --> ALPHA+
ALPHA    ::= LETTRE, CHIFFRE
LETTRE   ::= a, b, ..., z
CHIFFRE  ::= zéro, un, ..., neuf
```

Ex : id représente l'identificateur "toto".

```
t -- o -- t -- o
```

Cette façon de définir un identificateur nous semble peu satisfaisante car elle ne fait pas assez abstraction de la représentation concrète d'un identificateur. C'est pour la même raison que les notions d'identificateur et de constante sont représentées comme des symboles terminaux de la grammaire concrète hors-contexte du langage, et que deux identificateurs sont différenciés à l'aide d'un autre formalisme de nature lexicographique. Dans un analyseur syntaxique, un identificateur est donc considéré comme un terminal, et une information supplémentaire différencie les identificateurs.

Dans un arbre abstrait, identificateurs et constantes sont donc représentés par des feuilles, décorées par des annotations lexicales qui mémorisent la chaîne de caractères associée. Par exemple, l'identificateur "toto" est représenté par : id - - - - "toto".

L'annotation lexicale "toto" est une décoration de l'opérateur id, que nous avons rattachée visuellement par des pointillés. Par abus de notation, nous pourrons par la suite alléger les figures en représentant l'identificateur "toto" par "l'opérateur" toto.

Nous pouvons maintenant définir la RI d'un programme comme l'arbre abstrait du programme décoré d'annotations diverses. La donnée de l'arbre et des annotations lexicales est suffisante pour reconstituer un texte source équivalent, et constitue donc le squelette de la RI. Les autres annotations permettent de mémoriser des informations de nature diverses comme les commentaires, la mise en page, les erreurs contextuelles et, de façon générale, toute information, déductible ou non du squelette, qu'un outil de l'atelier juge nécessaire de conserver. Le lecteur obtiendra des exemples d'annotations lors de l'exposé de la RI de l'atelier Adèle.

3.4. PROBLEMES LIES A LA CONCEPTION D'UNE GRAMMAIRE ABSTRAITE.

Lorsqu'un atelier est configuré pour un langage de programmation, la première tâche à effectuer consiste à élaborer la grammaire abstraite du langage considéré. Nous désignerons sous le terme de concepteur la ou les personnes qui effectuent ce travail.

Le concepteur est guidé dans ses choix par le manuel de référence du langage, qui en définit la syntaxe et la sémantique. En pratique, seule la syntaxe hors contexte est bien formalisée, alors que la syntaxe contextuelle et la sémantique sont le plus souvent décrites en langue naturelle. Le concepteur est donc amené à étudier la syntaxe concrète du langage pour en déduire la syntaxe abstraite. Il affine les premiers résultats à l'aide de la sémantique du langage et de sa propre vision sur l'utilisation de l'arbre abstrait dans l'atelier.

Tout laisse donc croire que différents concepteurs définiront différentes grammaires abstraites pour le même langage. C'est pour mettre en évidence des styles de conception que nous avons étudié quatre définitions de grammaires abstraites pour deux langages présentant quelques similitudes, Pascal et Ada. Pour Pascal existent Mentor et Adèle ; la définition formelle du langage Ada <Ada 80b> est centrée sur sa grammaire abstraite ; enfin, Diana <Goos 81> est une représentation intermédiaire de programmes Ada destinée à être le pivot d'un compilateur.

Cette étude montre que c'est l'idée de l'utilisation d'une RI par un atelier qui aboutit aux différences de conception. Le plus souvent, le concepteur décide (consciemment ou non) de favoriser un outil ou un groupe d'outils au détriment des autres. Une même notion du langage peut ainsi être fractionnée suivant son contexte d'utilisation afin de faciliter certains traitements, comme la notion d'identificateur. A l'opposé, des notions initialement distinctes, au moins par leur forme concrète, sont regroupées, permettant ainsi un traitement plus synthétique. Enfin, il est très tentant de "gommer" des incohérences et bizarreries rencontrées dans la définition du langage.

Peut-être n'y a-t-il pas de solution définitive à ce problème, tant que les concepteurs d'un langage ne définiront pas sa grammaire abstraite avant sa représentation concrète, comme c'est le cas pour la plupart des langages actuels. C'est pourquoi nous n'apportons aucune solution, mais nous contentons d'énumérer quelques problèmes, en laissant le lecteur seul juge.

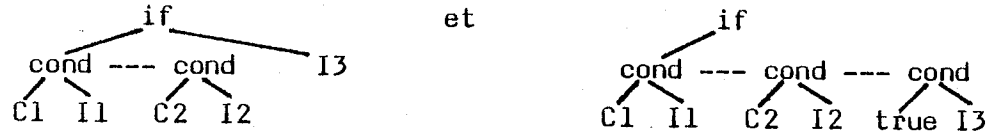
Les grammaires abstraites étudiées ne font pas appel à la notion de rameau, et emploient explicitement des opérateurs de liste. Nous nous sommes permis de présenter nos exemples sans ces opérateurs : le lecteur pourra reconstituer sans peine la définition originale.

3.4.1. Arbre abstrait et informations contextuelles.

La définition formelle du langage Ada fait appel aux notions d'arbre abstrait primitif et d'arbre abstrait étendu : l'arbre abstrait primitif est produit par un analyseur de syntaxe hors contexte, tandis que l'arbre abstrait étendu est le résultat d'un analyseur contextuel, qui a pu effectuer une normalisation des opérateurs.

Ex 1 : l'instruction : if C1 then I1 elsif C2 then I2 else I3 end if
est réécrite : if C1 then I1 elsif C2 then I2 elsif true then I3 end if

Les deux arbres correspondants sont :



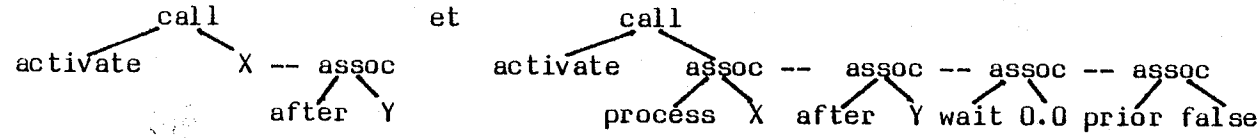
Ex 2 : Considérons la procédure définie au paragraphe 6.4.2 de <Ada 80b> :

```
procedure  activate (process : in process_name ;
                     after   : in process_name := no_process ;
                     wait    : in duration     := 0.0 ;
                     prior   : in boolean      := false) ;
```

L'instruction d'appel : activate (X, after ==> Y) est réécrite en :

activate (process ==> X, after ==> Y, wait ==> 0.0, prior ==> false).

Les deux arbres correspondants sont :



L'arbre abstrait étendu aboutit à des traitements sémantiques plus réguliers, avec moins de cas particuliers. Mais son choix en tant que RI impose aux utilisateurs d'un atelier d'écrire systématiquement des programmes standardisés (ou alors l'éditeur syntaxique devient complexe à écrire, ainsi que le décompilateur). Cette contrainte n'est pas forcément mauvaise, puisqu'elle aboutit à des normes de programmation. Mais elle fait perdre les facilités

d'écriture du langage, et de ce fait peut paraître trop contraignante aux utilisateurs d'un éditeur syntaxique.

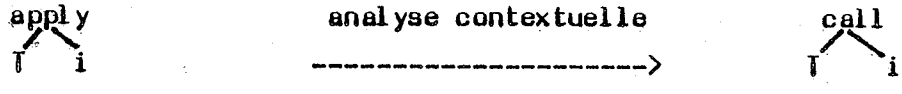
Nous pensons qu'il faut laisser ces facilités aux programmeurs, et donc choisir l'arbre abstrait primitif comme RI dans un atelier. Les normes de programmation peuvent alors être assurées par une commande de normalisation.

La conséquence de ce choix est la simplification des outils de construction d'un programme, qui n'ont pas besoin d'effectuer des traitements contextuels avant de pouvoir construire l'arbre d'un programme. Une autre conséquence est l'alourdissement des algorithmes traitant la sémantique d'un programme ; mais ce problème peut être minimisé par l'emploi judicieux d'annotations contextuelles (dans l'exemple 2, l'opérateur "call" peut être annoté par les valeurs initiales des paramètres implicites).

3.4.2. Ambiguïtés syntaxiques.

Certaines constructions du langage, que l'on veut différencier d'un point de vue sémantique, ont la même représentation textuelle : une analyse contextuelle est alors nécessaire pour lever l'ambiguïté. Faut-il alors définir un seul opérateur commun aux différentes constructions, ou au contraire définir un opérateur par construction ? Le problème rejoint celui du paragraphe précédent et sa solution est donc identique.

Considérons par exemple l'expression Ada : "T(i)". Cette construction peut désigner un élément de tableau, un appel de fonction, ou un aggrégat. Puisque seule une analyse contextuelle peut effectuer la distinction, l'arbre abstrait primitif contient un seul opérateur pour désigner les trois notions, l'opérateur "apply". Cet opérateur est transformé de manière adéquate dans l'arbre abstrait étendu, suivant le contexte.



L'utilisation de cet opérateur "apply" dans un éditeur syntaxique peut paraître peu naturelle à un programmeur, qui conçoit différemment élément de tableau, appel de fonction et aggrégat, et désire manipuler ces notions différemment : il préfère donc trois opérateurs au lieu d'un seul. Si le concepteur de l'arbre abstrait effectue ce choix, il en résulte une complexité

plus grande de l'analyseur syntaxique : l'analyse hors contexte et l'analyse contextuelle d'un programme ne peuvent plus être considérées comme indépendantes ; et l'analyseur contextuel ne peut pas travailler sur l'arbre abstrait, puisqu'il n'est pas encore créé : comment, dans ce cas, écrire un éditeur syntaxique effectuant des vérifications contextuelles sur l'arbre ?

Une première solution au problème consiste alors à définir les quatre opérateurs ("apply" et les trois opérateurs différenciés) dans la grammaire abstraite. L'opérateur "apply" est à usage interne de l'atelier et n'est pas connu de ses utilisateurs. Mais cette solution n'est pas très satisfaisante dans la mesure où la même construction du langage peut être représentée (au moins temporairement) de plusieurs manières distinctes, ce qui contredit le principe de l'unicité de la représentation.

C'est pourquoi nous préférons définir un seul opérateur "apply" dans la grammaire abstraite. L'utilisateur de l'éditeur syntaxique peut être satisfait s'il peut utiliser trois noms synonymes pour désigner cet opérateur : élément de tableau, appel de fonction et aggrégat. De cette manière, l'unicité de représentation est préservée sans défavoriser l'utilisateur de l'atelier. Là encore, des annotations judicieusement choisies facilitent les traitements sémantiques.

Ce problème se trouve aussi dans Pascal : dans une expression, un identificateur peut désigner l'accès à une variable ou l'appel d'une fonction sans paramètres. La solution proposée ne marche pas car l'opérateur appel de fonction possède syntaxiquement deux fils, alors qu'un identificateur est une feuille de l'arbre. Par exemple, l'expression F peut avoir deux formes, suivant le contexte : F et appel-fonction .

En l'absence de meilleure solution, nous avons choisi dans Adèle de permettre les deux représentations. L'analyseur contextuel, mis en présence de la première forme, transforme l'arbre pour le mettre sous la deuxième forme. C'est d'ailleurs la seule transformation effectuée par l'analyseur contextuel.

3.4.3. Exemples divers.

(a) Identificateurs.

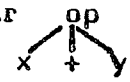
Diana distingue les identificateurs suivant la nature de leur utilisation ; on y trouve donc les opérateurs : identificateur de variable, de type, de procédure, ..., et utilisation d'identificateur. Cette distinction se justifie dans cette RI par la contrainte que les annotations sont définies "hors contexte". La donnée d'un opérateur est suffisante pour déterminer ses annotations : un identificateur de type possède l'annotation "type dénoté" ; un identificateur de variable possède de plus l'annotation "protection en écriture" ; une utilisation d'identificateur possède la seule annotation "lien vers la déclaration". Cette solution présente l'inconvénient d'apporter des informations contextuelles dans l'arbre abstrait, et impose aux utilisateurs de l'éditeur une vision complexe de la notion d'identificateur.

Nous présentons les annotations au chapitre 3.5.4, comme définies indépendamment des opérateurs. Leur présence sur un noeud de l'arbre peut être fonction du contexte dans lequel est plongé le noeud. Dans ces conditions, il est possible de ne définir qu'un seul opérateur identificateur, annoté selon son contexte.

(b) Opérateurs arithmétiques.

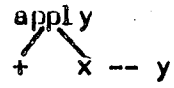
Considérons l'expression "x + y". Dans la mesure où le langage le permet, comme c'est le cas en Pascal, il est préférable de la représenter par l'arbre

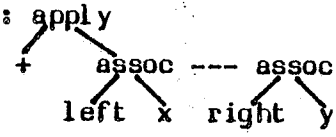
plutôt que par



; l'arbre est plus concis et plus naturel.

Par contre, en Ada, l'utilisateur a la possibilité de redéfinir l'addition. L'arbre est donc : apply et l'arbre abstrait est : apply





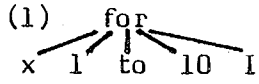
Nous pensons que même en Ada il est possible d'utiliser l'opérateur "+", en l'annotant quand il correspond à une fonction définie dans le programme.

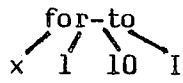
(c) Instruction pour.

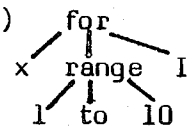
Considérons l'instruction "pour" de Pascal :

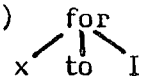
for x := 1 to 10 do I (ou for x := 10 downto 1 do I).

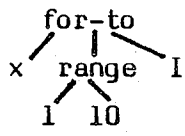
Examinons différentes manières de la représenter :

(1) 

(2) 

(3) 

(4) 

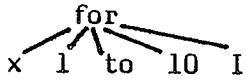
(5) 

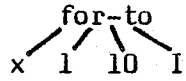
, ...

Le cas (2) est tiré d'Adèle, (4) de Mentor, et (5) de la définition formelle d'Ada.

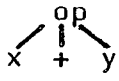
Conceptuellement, l'opérateur "range" des cas (3) et (5) n'apparaît pas comme très utile, puisque c'est le seul opérateur utilisable à cet emplacement, qu'il est obligatoire, et qu'il n'est employé nulle part ailleurs. Les cas (3) et (5) peuvent donc se ramener respectivement aux cas (1) et (2). De même, le cas (4) se ramène au cas (1).

Il ne nous reste plus qu'à choisir entre :

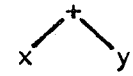
(1) 

et (2) 

Ces deux cas ressemblent fort à l'exemple précédent :



et

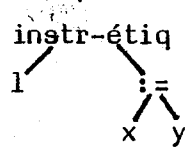


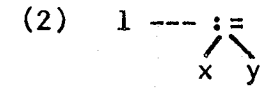
Cette analogie nous a fait choisir le cas (2) pour la définition d'Adèle.

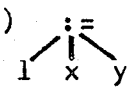
Remarque : les choix effectués pour Mentor et Ada sont dus à la volonté des concepteurs de ne pas définir d'opérateurs ayant une arité plus grande que trois. Cette contrainte peut sembler a priori bonne, car elle permet de définir des opérateurs plus simples ; mais elle devrait s'appliquer lors de l'élaboration d'un nouveau langage, et non pas pour un langage déjà existant.

(d) Etiquettes.

Considérons le traitement des étiquettes en Pascal : "1 : x := y". Cette instruction peut être représentée de trois manières :

(1) 

(2) 

(3) 

Le cas (2) présente une instruction étiquetée comme deux instructions, et ne correspond donc pas au langage. Le cas (3) considère les étiquettes comme un champ optionnel de toutes les instructions, et est plus simple que le cas (1), car la grammaire abstraite nécessite un opérateur de moins.

3.4.4. Conclusion : quelques conseils utiles.

La définition d'une grammaire abstraite est une tâche lourde de conséquences : dans un atelier paramètré par le langage de programmation, il existe une relation directe entre les opérateurs de la grammaire et les spécications externes de l'éditeur syntaxique. Toute anomalie de conception se traduit donc pour les utilisateurs comme une aberration de l'atelier, qui, au mieux choque leur propre intuition d'un programme, et au pire leur fait refuser l'utilisation de l'atelier.

Nous ne pensons pas qu'il soit possible de déduire automatiquement la grammaire abstraite d'un langage à partir de sa grammaire concrète, et d'ailleurs ce ne serait pas souhaitable : la définition d'un langage devrait commencer par la conception de sa grammaire abstraite, avant celle de sa (ou ses) grammaire(s) concrète(s).

Nous ne pouvons donc que proposer quelques conseils permettant d'éviter la définition de grammaires difficilement exploitables.

Ne pas inclure d'informations contextuelles dans l'arbre abstrait : utiliser les annotations pour définir ce type d'informations. En particulier, ne pas lever les ambiguïtés syntaxiques.

Respecter le principe d'unicité de la représentation d'un concept, comme par exemple la notion d'identificateur.

Utiliser des solutions homogènes pour des problèmes analogues, comme les opérateurs arithmétiques et l'instruction "pour".

Choisir des solutions les plus "simples" possibles, par exemple en minimisant le nombre d'opérateurs (penser que l'utilisateur d'un éditeur syntaxique doit connaître tous les opérateurs). En particulier, ne pas définir d'opérateur "liste" inutile, mais utiliser la notion de rameau.

Nous espérons que l'utilisation de ces conseils permettra de concevoir des grammaires abstraites de qualité, facilement utilisables dans un atelier.

3.5. OPERATIONS SUR UNE RI.

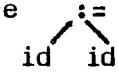
Ce paragraphe a pour but de décrire les principales opérations qu'il est possible d'effectuer sur la RI d'un programme. Pour pouvoir manipuler une RI, il est nécessaire de désigner sans ambiguïté un arbre ou un sous-arbre. Nous décrivons ensuite les principales opérations de consultation, de modification, de création d'un arbre et d'annotations sur un arbre, puis une opération de transformation d'arbre utilisant un mécanisme de filtrage. Cette liste n'est pas exhaustive, mais elle donne une bonne idée des opérations possibles.

La plupart des opérations que nous définissons sont le reflet direct des commandes d'un éditeur syntaxique. Seules les opérations de nature textuelle ne se retrouvent pas dans ce paragraphe ; mais elles se déduisent facilement des primitives de base. Par exemple, les déplacements textuels sont des parcours des feuilles de l'arbre.

Pour ne pas dépendre d'un quelconque langage de programmation et de ses limitations éventuelles, nous avons adopté un formalisme pseudo-mathématique pour décrire les fonctions, facilement traductibles dans un langage donné.

3.5.1. Arbres et emplacements d'arbres.

D'un point de vue théorique, il est nécessaire de distinguer arbre et occurrence d'arbre. Par exemple, l'arbre



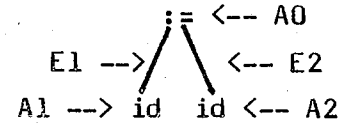
est composé de deux occurrences du même arbre id.

Par abus de langage, nous parlerons d'arbre pour désigner une occurrence d'arbre. Ceci ne pose pas de problèmes, sauf sur la notion d'identité.
On notera :

A1 = A2 si A1 et A2 désignent la même occurrence d'arbre,
A1 ≃ A2 si A1 et A2 désignent deux occurrences du même arbre.

On a : A1 = A2 ==> A1 ≃ A2.

Une notion supplémentaire peut être utile : l'emplacement. On peut définir intuitivement un emplacement comme la branche qui relie deux arbres.
Ex : l'arbre A0 est composé de deux emplacements E1 et E2, contenant respectivement les arbres A1 et A2 :



Remarque : Mentor utilise exclusivement des repères d'emplacements, et Adèle des repères d'arbres.

Nous supposerons que l'implémentation permet de retrouver quel est l'arbre contenu dans un emplacement, et dans quel emplacement est contenu un arbre (une façon simple est de conserver un lien de chaînage entre un arbre et son père). Sous cette condition, arbre et emplacement sont des notions équivalentes pour presque toutes les opérations. Seules les opérations de modification d'un arbre permettent de distinguer ces deux notions.

Ex :

Quand on remplace l'arbre A2 par l'arbre A3, on obtient :

L'emplacement E2 ne contient donc plus l'arbre A2. L'équivalence initiale entre E2 et A2 n'est plus respectée.

La notion d'emplacement peut être utile à certains outils de l'atelier : un éditeur permet aux utilisateurs des modifications de l'endroit courant ; un interpréteur contrôle l'emplacement à exécuter : si l'utilisateur arrête l'interpréteur, modifie l'emplacement courant, puis relance l'interpréteur, alors l'exécution reprend sur le nouvel arbre et non pas sur l'ancien.

Nous pensons qu'il est suffisant de définir les opérations avec la seule notion d'arbre, pourvu que la transformation arbre-emplacement soit assurée. L'expérience montre en effet que l'implémentation des opérations est bien plus efficace quand elles sont définies sur des arbres (repérées par exemple par un pointeur) plutôt que sur des emplacements (repérés par exemple par un pointeur -arbre père- et un entier -rang de l'emplacement-). Un outil n'aura besoin de convertir un repère d'arbre en repère d'emplacement que lorsqu'il sera dans un état stable, où l'arbre pourra être modifié sans risque par lui-même ou par un autre outil.

Signalons enfin l'impossibilité de modifier l'opérateur associé au noeud racine d'un arbre, sans modifier le repère de ce noeud. Autrement dit, cette restriction assure que deux arbres distincts (au sens strict) ont des repères distincts.

3.5.2. Opérations de consultation d'un arbre.

Soient A l'ensemble des repères d'arbres, E l'ensemble des repères d'emplacements, O l'ensemble des opérateurs, P l'ensemble des phylums. Les principales opérations de consultation sont les suivantes :

identité (=)	: A x A	--> {vrai, faux}
équivalence (≡)	: A x A	--> {vrai, faux}
conversion d'arbre en emplacement (AE)	: A	--> E
conversion d'emplacement en arbre (EA)	: E	--> A

(on a bien sûr : EA (AE (A)) = A et AE (EA (E)) = E)

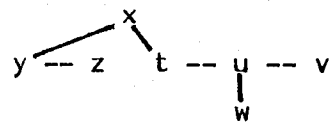
nature d'un arbre (oper)	: A	--> O
arbre indéfini (Λ)	: O	--> A
fils (fils)	: A x N	--> A

fils (a,n) retourne le premier élément du n-ième rameau de a.

élément suivant d'un rameau (suiv)	: A	--> A
élément précédent d'un rameau (prec)	: A	--> A
premier élément d'un rameau (prem)	: A	--> A
dernier élément d'un rameau (dern)	: A	--> A
père d'un arbre (père)	: A	--> A
rang du rameau contenant un arbre (rang)	: A	--> N
position d'un arbre dans un rameau (pos)	: A	--> N

Remarque : d'autres primitives de navigation dans l'arbre peuvent être définies, comme le parcours préfixé ou postfixé, le passage à la racine, etc. Nous ne les décrivons pas ici, car elles se déduisent facilement de ce groupe de primitives.

Ex :



fils (x,1) = y fils (x,2) = t fils (x,n) = Λ $\forall n \neq 1,2$

père (y) = père (z) = père (t) = père (n) = père (v) = x

père (x) = Λ

suiv (y) = z suiv (z) = Λ

prec (z) = y prec (y) = Λ

prem (t) = prem (u) = prem (v) = t

dern (t) = dern (u) = dern (v) = v

rang (t) = rang (u) = rang (v) = 2

pos (t) = 1 pos (u) = 2 pos (v) = 3

3.5.3. Opérations de création et de modification d'un arbre.

création (créer) : 0 --> A

créer (0) retourne un repère d'arbre non encore utilisé. L'arbre retourné est composé du nombre nécessaire de rameaux fils, chacun de ces rameaux étant réduit à l'arbre créer (vide).

En pratique, il est utile de pouvoir créer et relier en même temps un arbre et ses fils (qui ont été créés auparavant). Nous définissons donc des fonctions créer_i ($i \in N$) permettant cette opération pour des opérateurs d'arité i. Pour cela, nous supposons l'existence d'un opérateur seq (défini implicitement dans toute grammaire abstraite par la règle seq ==> P*). Cet opérateur permet de désigner par un seul repère d'arbre l'ensemble des arbres devant constituer un futur rameau de l'arbre à créer. Les fonctions créer_i sont donc :

créer0 : 0 --> A (opérateur nullaire)

créer1 : 0 x A --> A (opérateur unaire)

créer1 (0, a) retourne 0 (a) si oper(a) ≠ seq

et 0 (a1--a2--...--an) si a = seq (a1--a2--...--an)

créer2 : 0 x A x A --> A

.....

destruction (détruire) : A --> Ø

Cette procédure est inutile si l'implémentation dispose d'un "ramasse-miettes".

remplacement (remplacer) : A x A --> Ø

remplacer (a1, a2) remplit AE(a1) par a2.

extraction (oter) : A --> Ø

oter(a) extrait a de son rameau (et le remplace par créer(vide) si le rameau ne contient que a).

insertion : avant : A x A --> Ø

après : A x A --> Ø

avant (a1, a2) insère a1 avant a2 ;

après (a1, a2) insère a1 après a2.

Pour ces deux primitives : si a2 = Λ , l'opération n'a pas d'effet.

Si a2 = seq , il y a insertion de b1--b2--...--bn.
b1--b2--...--bn

recopie (copie) : A --> A

On a : copie (a) $\neq$ a et copie (a) $\simeq$ a.

Opérations sur des sous-listes : il est souvent intéressant de pouvoir manipuler une partie d'un rameau, ou sous-liste. Une sous-liste est désignée par deux repères d'arbres, indiquant le premier et le dernier élément de la sous-liste. Les opérations suivantes n'ont pas d'effet si les deux repères désignent des arbres n'appartenant pas au même rameau.

détruireliste : A x A --> $\emptyset$

oterliste : A x A --> A

oterliste (a1, a2) retourne seq
a1 --- ... --- a2

remplacerliste : A x A x A --> $\emptyset$

remplacerliste (a1, a2, a3) ajoute a3 après a2 et extrait a1 -- ... -- a2.

Si a3 = seq , il y a rajout de b1 -- ... -- bn.
b1 -- ... -- bn

copieliste : A x A --> $\emptyset$

copieliste (a1, a2) = seq
copie(a1) -- ... -- copie(a2)

3.5.4. Opérations sur les annotations.

Une annotation peut représenter n'importe quel type de données : entier, chaîne de caractères, arbre, etc. (l'implémentation peut restreindre le domaine de types), et possède un repère pour la désigner. Pour chaque type, l'implémentation fournit des primitives de consultation et modification.

L'expérience montre que la présence d'une annotation sur un arbre dépend du contexte d'utilisation de cet arbre. Par exemple, une déclaration d'identificateur possède l'annotation "type dénoté", tandis qu'une utilisation d'identificateur possède l'annotation "lien vers déclaration". Cet exemple montre aussi qu'une annotation possède un nom indépendant de son repère, qui désigne un moyen d'accéder à la valeur de l'annotation.

```
créer_annotation      : nom x A      --> annotation
existe_annotation     : nom x A      -->  vrai, faux
trouver_annotation    : nom x A      --> annotation
détruire_annotation   : annotation   --> 0
trouver_nom           : annotation   --> nom
lister_annotations    : A            -->  (annotation)
```

3.5.5. Opération d'instanciation par filtrage.

Nous désirons définir une primitive permettant la recherche d'un arbre ayant un certain "schéma" (filtrage), afin de créer un nouvel arbre à partir de l'arbre trouvé (instanciation). Cette primitive permet un mécanisme primitif de réécriture.

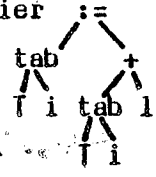
Un schéma est un arbre dont certaines feuilles sont appelées variables d'arbre. Un schéma S filtre un arbre A si l'on peut trouver une correspondance entre les variables de S et des sous-arbres de A permettant d'identifier S à A en remplaçant les variables de S par les sous-arbres de A.

Par exemple, si $x et $y sont des variables d'arbres, le schéma S filtre l'arbre A :

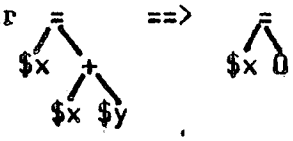
En termes de représentation concrète, le schéma "$x := $x + $y" filtre l'instruction "T[i] := T[i] + 1".

Instancier un arbre A par la règle de réécriture S1 ==> S2 consiste à filtrer A par S1, puis à créer un nouvel arbre obtenu en remplaçant les variables de S2 par les arbres correspondant aux variables de S1.

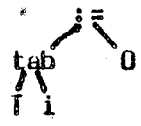
Par exemple, instancier :=



par = ==> =



crée l'arbre :



Nous définissons une annotation particulière, "variable", de type entier, qui, associée à l'opérateur vide, permet de jouer le rôle de variable d'arbre. A chaque valeur de cette annotation correspond une variable distincte.

filtrer : A x A --> vrai, faux

filtrer (a,S) indique si l'arbre a correspond au schéma S.

instancier : A x A x A --> A

instancier (a, S1, S2) retourne l'arbre obtenu à partir de a par la règle de réécriture S1 ==> S2.

(si filtrer (a, S1) = faux, alors instancier (a, S1, S2) =Λ).

3.6. UTILISATION D'UNE RI DANS UN ATELIER.

Dans les chapitres précédents, nous avons présenté les mécanismes de base permettant d'écrire un outil de manipulation de programmes. Mais ces concepts ne décrivent pas comment un outil s'intègre à un atelier paramètré par le langage de programmation. C'est pourquoi ce paragraphe essaie d'adopter une solution à deux problèmes :

- le "dialogue" entre différents outils d'un atelier,
- la génération "automatique" d'outils pour un langage donné.

Sur ces deux points, il n'y a pas actuellement de réponse satisfaisante. C'est pourquoi nous nous contentons d'apporter des solutions partielles, fondées sur les expériences de Mentor, CPS et Gandalf.

Les différents outils d'un atelier ne peuvent pas être considérés comme indépendants : ce sont plutôt des activités s'exécutant en coroutines. Par exemple, éditeur et décompilateur sont nécessairement liés, puisque le premier rend compte au deuxième des modifications apportées au programme, afin d'en mettre à jour l'affichage sur l'écran. De même l'interpréteur est concerné par une modification, qui peut annuler ou modifier le résultat partiel de l'exécution du programme. Une première solution à ce problème consiste à faire connaître à un outil l'ensemble des autres outils concernés par son action ; mais elle présente l'inconvénient de remettre en cause les outils existants lors de l'introduction d'un nouvel outil dans l'atelier. C'est pourquoi nous préférons un mécanisme de déroutement similaire à celui défini dans Gandalf ("action routines").

Un des buts poursuivis par les concepteurs d'un atelier est de pouvoir adapter facilement l'atelier à un langage de programmation. Actuellement, seule la syntaxe hors-contexte d'un langage fait facilement l'objet de traitements paramétrés. Nous illustrons ceci par l'étude de la décompilation et de l'analyse syntaxique.

3.6.1. Déroutements à l'intérieur des opérations sur la RI.

La plupart des calculateurs disposent d'un mécanisme de déroutement d'un programme, qui déclenche l'exécution d'un autre programme lors de la rencontre d'évènements particuliers ("exceptions") : tentative de division par zéro, de lecture d'une information protégée, etc. Le plus souvent, une exception empêche

la reprise du programme interrompu, mais il arrive que le déroutement se comporte comme un appel de procédure imprévu : c'est le cas de la "faute de page" dans un système à mémoire virtuelle. Certains langages évolués, comme PL/1 et Ada, fournissent aux programmeurs les moyens de traiter une exception.

Le système Gandalf utilise un tel mécanisme dans son noyau de manipulation d'arbres. Initialement, ceci avait pour but de vérifier la syntaxe contextuelle des programmes au fur et à mesure de leur édition. Mais les concepteurs du système se sont vite rendu compte que ce mécanisme avait de nombreuses applications, et facilitait la communication entre les différents outils de l'environnement de programmation.

Les différentes activités du noyau de gestion d'arbres sont classifiées en types d'actions élémentaires comprenant, par exemple, création, insertion, destruction, positionnement et départ d'un noeud de l'arbre, etc. Pour chacune de ces actions de base, il y a appel à une procédure de déroutement, dont les paramètres sont le repère du noeud concerné et le type d'action effectuée. Le déroutement s'effectue soit avant, soit après l'action de base, suivant son type (par exemple l'appel s'effectue après l'action de création et avant l'action de destruction), et retourne une valeur indiquant s'il faut annuler l'action, si le noeud concerné a été modifié, etc. Le programme de déroutement effectue lui-même des actions sur l'arbre, ce qui peut bien sûr provoquer de nouveaux déroutements.

Ce mécanisme apporte le grand avantage de rendre modulaire la communication entre outils. Imaginer par exemple l'existence d'un outil de vérification de la syntaxe contextuelle d'un programme, devant être activé lors de chaque modification du programme. Par le biais des déroutements, on peut assurer son activation automatique par tous les autres outils de l'atelier, qui ne connaissent même pas son existence. De plus, l'ajout d'un nouvel outil ne remettra pas en cause ce principe.

Ce mécanisme peut être décrit de la manière suivante : quand un outil devient actif, il se "présente" au noyau de gestion de la RI. Cette déclaration de présence s'accompagne de la liste des déroutements à activer pour le compte de l'outil. Ceci permet d'une part de définir le noyau indépendamment des outils, et d'autre part d'adapter les traitements effectués sur un programme à l'état général de l'atelier.

Il est possible d'envisager des implémentations moins dynamiques. Par exemple dans Gandalf, les déroutements sont décrits à la génération d'un atelier pour un langage et une classe d'outils donnés.

Un prolongement intéressant consisterait à étendre ce mécanisme à la manipulation des annotations. Une application immédiate serait la communication entre un analyseur de syntaxe contextuelle, qui détecte et marque les arbres erronés, et le décompilateur, qui met en évidence ces erreurs sur l'écran, sans que ces deux outils aient connaissance l'un de l'autre.

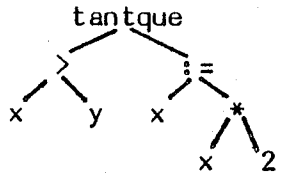
3.6.2. Description d'un décompilateur.

Un décompilateur permet la visualisation de l'arbre sous une forme compréhensible par l'être humain. Le plus souvent, cette forme est celle d'un texte source "bien" présenté ("pretty printing").

Le décompilateur doit effectuer deux tâches : traduire un arbre en termes concrets, et mettre à jour l'écran au fur et à mesure des opérations d'édition. Cette deuxième tâche peut être modélisée comme la mise à jour d'annotations sur l'arbre, mise en oeuvre à l'aide de procédures de déroutement. Ces annotations sont des repères vers une structure d'affichage (dans Adèle, l'arbre de boîtes).

L'opération de visualisation de l'arbre peut être entièrement dirigée par la syntaxe abstraite du langage. A chaque opérateur correspond un schéma de décompilation, comportant des indications de mise en page. L'interprétation d'un schéma met en oeuvre un parcours récursif descendant de l'arbre, et fait appel aux schémas des opérateurs descendants de la racine.

Par exemple, le schéma de l'opérateur Pascal "tantque" peut être :
écrire "while",
décompiler le premier fils, ou écrire " expression " s'il est vide,
écrire "do",
passer à la ligne et augmenter l'indentation de 3 blancs,
décompiler le deuxième fils, ou écrire " statement " s'il est vide,
passer à la ligne et diminuer l'indentation de 3 blancs.

L'arbre : 

s'écrit : while x > y do
x := x * 2

Le système Gandalf définit un schéma de décompilation à l'aide d'une chaîne de caractères composée de texte à imprimer et de commandes (commençant par le caractère spécial %). Parmi ces commandes, on trouve :

%1, %2, ... : décompiler le premier, le deuxième, ... fils,
%n : sauter à la ligne,
%+ : augmenter l'indentation,
%- : diminuer l'indentation.

Dans ce système, la décompilation de l'opérateur "tantque" s'exprime par le schéma : while %1 do %+ %2 %- %n

D'autres commandes permettent le parenthésage correct des expressions arithmétiques suivant la priorité des opérateurs, la spécification d'un texte séparateur à imprimer entre chacun des éléments d'une liste, etc.

Le cas des emplacements vides est résolu dans Gandalf par le fait qu'il faut définir pour chaque phylum un opérateur spécial ("metanode") jouant le rôle de notre opérateur vide : le schéma de l'opérateur correspondant à une expression vide est "expression", par exemple. Par contre, il n'y a pas dans ce système de notion d'emplacement optionnel. Par exemple, il y a toujours affichage de "else" lors de la décompilation d'une instruction conditionnelle.

Nous n'avons pas approfondi les problèmes que pose la transposition des schémas de décompilation à notre formalisme d'arbre. La notion de rameau impose de pouvoir spécifier un séparateur pour chacun des rameaux fils d'un arbre ; l'unicité de l'opérateur vide impose un schéma de décompilation défini sur un phylum (le phylum EXPRESSION, par exemple, a le schéma "expression"). Enfin, il faut pouvoir prendre en compte les annotations, pour la décompilation des emplacements optionnels (qui sont représentés par l'opérateur vide annoté).

Indépendamment du formalisme, il serait intéressant de pouvoir spécifier plusieurs schémas de décompilation pour un opérateur : chaque schéma serait appliqué selon le contexte, c'est-à-dire qu'il y aurait filtrage de l'arbre à décompiler afin de déterminer le schéma à appliquer. Cette idée intéressante fait l'objet de recherches de la part des concepteurs de Mentor.

3.6.3. Description d'un analyseur syntaxique.

Un analyseur syntaxique, produisant l'arbre abstrait d'un programme à partir de sa représentation textuelle, est utile dans un atelier pour au moins deux raisons :

- introduction dans l'atelier de programmes déjà existants,
- définition d'un éditeur mixte, permettant une construction syntaxique ou textuelle de fragments de programmes.

Il est devenu classique d'utiliser des outils générateurs d'analyseurs syntaxiques, ascendants ou descendants, admettant en entrée une grammaire hors-contexte augmentée "d'actions de compilation" généralement écrites à la main. Ces actions sont destinées à produire une représentation intermédiaire du programme analysé.

Il est possible de définir un langage d'expression des actions de compilation, permettant la production d'un arbre abstrait. Nous illustrons cette idée par l'exposé du système Métal, qui est composé d'un préprocesseur du générateur d'analyseurs ascendants SYNTAX. Ce préprocesseur admet comme actions des fonctions de création d'arbres <Mélèse 82>.

A chaque règle de la grammaire hors-contexte correspond une fonction de création d'arbre, paramétrée par les symboles non-terminaux de la règle, et retournant un arbre associé au non-terminal partie gauche de la règle. Lorsque ce non-terminal est utilisé en partie droite de règle, et donc est un paramètre d'une autre fonction, il y a remplacement du paramètre par l'arbre associé. Il y a donc création et greffe des sous-arbres jusqu'à la fin de l'analyse : l'arbre associé à l'axiome de la grammaire est l'arbre complet du programme.

Soit par exemple la règle de production de l'instruction Pascal "tantque" :
INST ::= while CONDITION do INST tantque (CONDITION, INST2).
L'action associée consiste à créer le noeud "tantque" ; le premier fils est l'arbre associé à CONDITION et le deuxième fils celui associé à INST2 (si un symbole est utilisé plusieurs fois dans la même règle, ses occurrences sont numérotées dans leur ordre d'apparition).

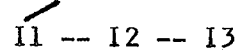
Les terminaux de la grammaire représentant un élément lexical (identificateurs et constantes) sont désignés dans les actions par des symboles spéciaux, indiquant la création d'un opérateur nullaire et d'une annotation lexicale.
Ex : EXP ::= identificateur ident (% id)

Deux actions, "pre" et "post" permettent la construction d'un arbre de type liste : pre (x, y) : insère x en tête de la liste y et retourne y,
post(y, x) : insère x en queue de la liste y et retourne y.

Ex : l'analyse ascendante d'une instruction composée Pascal fait appel aux règles et actions :

(1) INST ::= begin LINST end	LINST
(2) LINST ::= INST	lstat(INST)
(3) LINST ::= LINST ; INST	post(LINST2, INST)

Par exemple, le texte : "begin I1 ; I2 ; I3 end" (où I1, I2, I3 désignent des instructions quelconques) produit les réductions des règles (et donc les appels aux actions) successives : (2) (3) (3) (1).
L'arbre produit est donc :



<Mélèse 82> définit deux actions, "let" et "case", permettant de remédier aux grammaires "mal" écrites. En effet, la technique d'analyse utilisée, ascendante ou descendante, impose souvent une forme particulière à la grammaire hors-contexte, qui ne peut donc pas être aussi "élégante" que son concepteur le voudrait. Pour illustrer ce problème, nous utilisons un exemple tiré de l'article ci-dessus : dans cet exemple, trois grammaires produisent le même langage "Flip" de définition de transparents. Mais la façon d'écrire ces grammaires influe sur les actions à effectuer :

G1 : FLIP ::= en COULEUR écrire FLIP	fonte (COULEUR, FLIP)
FLIP ::= en COULEUR encadrer FLIP fincadre	cadre (COULEUR, FLIP)

Cette grammaire ne pose pas de problèmes particuliers.

G2 : TEXTE-OU-CADRE ::= écrire FLIP	fonte (vide, FLIP)
TEXTE-OU-CADRE ::= encadrer FLIP fincadre	cadre (vide, FLIP)

```
FLIP ::= en COULEUR TEXTE-OU-CADRE
            case  TEXTE-OU-CADRE
            when  fonte (X, Y) ==> fonte (COULEUR, Y)
            when  cadre (X, Y) ==> cadre (COULEUR, Y)
            end case
```

L'écriture de la règle FLIP empêche de savoir simplement s'il faut créer l'opérateur cadre ou fonte. C'est donc l'arbre associé à TEXTE-OU-CADRE qui permet de résoudre le problème.

L'action "case" effectue un filtrage de l'arbre TEXTE-OU-CADRE, par l'arbre fonte (X, Y) (où X et Y sont des variables d'arbre), puis, en cas d'échec, par l'arbre cadre (X, Y). Le succès du filtrage par fonte (X, Y) entraîne la création de l'arbre fonte (COULEUR, Y). L'arbre repéré par la variable X est perdu (il est créé uniquement pour respecter l'arité de l'opérateur fonte).

```
G3 : EN-COULEUR ::= en COULEUR                          fonte (COULEUR, vide)
     FLIP ::= EN-COULEUR écrire FLIP
                case EN-COULEUR
                when fonte (X, Y) ==> fonte (X, FLIP)
                end case
     FLIP ::= EN-COULEUR encadrer FLIP fincadre
                case EN-COULEUR
                when fonte (X, Y) ==> cadre (X, FLIP)
                end case
```

Au moment de l'utilisation de la règle EN-COULEUR, il n'est pas possible de savoir s'il faut créer un opérateur fonte ou cadre. Il y a donc création artificielle de l'opérateur fonte. L'utilisation de la règle FLIP adéquate permet de créer l'opérateur adéquat.

L'instruction "let" permet d'écrire plus clairement une instruction "case", lorsqu'il n'y a qu'un seul choix :

case a when b ==> c end case peut être écrit : let b = a in c.

On peut donc réécrire G3 :

```
G3 : EN-COULEUR ::= en couleur                  fonte (COULEUR, vide)
     FLIP ::= EN-COULEUR écrire FLIP            let fonte (X,Y) = EN-COULEUR
                                                in  fonte (X, FLIP)
     FLIP ::= EN-COULEUR encadrer FLIP fincadre let fonte (X,Y) = EN-COULEUR
                                                in  cadre (X, FLIP)
```

L'ensemble des actions de création d'arbre énumérées dans ce paragraphe permet la construction d'un analyseur syntaxique ascendant. Une étude serait nécessaire pour l'adapter à l'analyse descendante d'une part, et à la notion de rameau d'autre part.

3.7. RI DANS L'ATELIER ADELE.

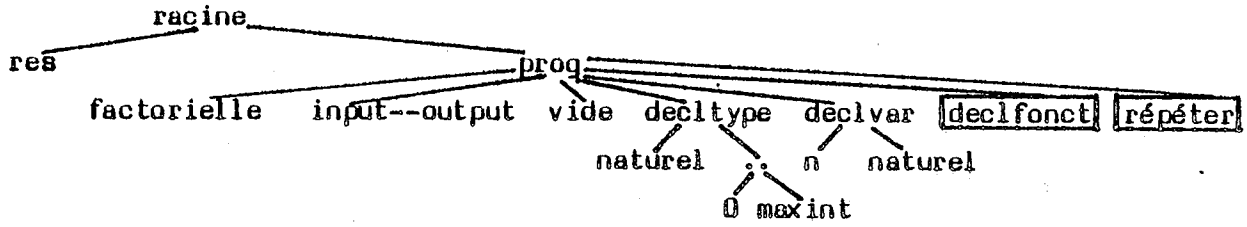
3.7.1. Grammaire abstraite du langage Pascal.

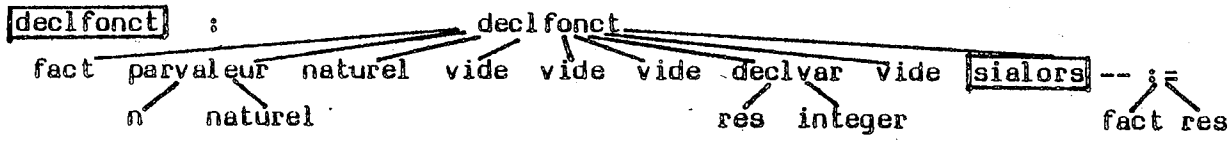
La grammaire abstraite de Pascal définie dans l'atelier Adèle est donnée à l'annexe A2. Aussi, nous nous contentons de l'illustrer en présentant l'arbre du programme suivant :

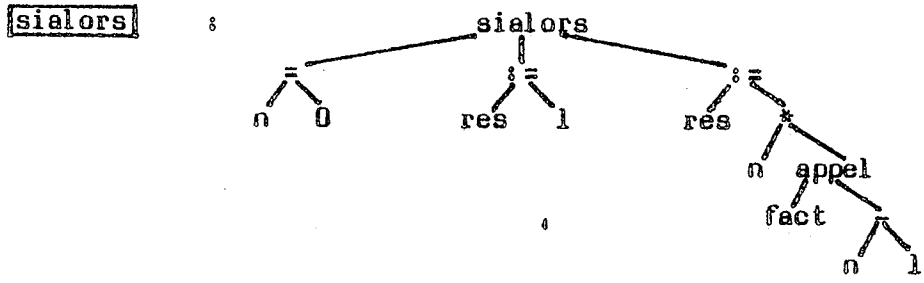
```
program factorielle (input, output) ;
    type naturel = 0 .. maxint ;
    var n : integer ;
    function fact (n : naturel) : naturel ;
       var res : integer ;
       begin
         if n=0 then res := 1 else res := n*fact(n-1) ;
         fact := res
       end ;

    begin
       repeat
         writeln ('rentrez un nombre') ;
         read (n) ;
         res := 0 ;
         if n < 0
         then writeln ('SVP rentrez un nombre positif')
         else writeln ('la factorielle est', fact(n))
       until n=0
    end.
```

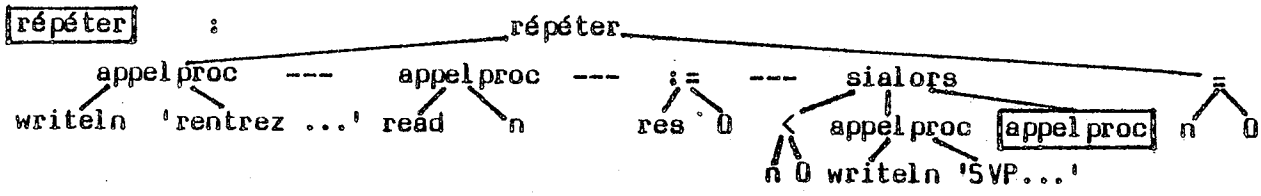
Ce programme est incorrect : l'instruction "res := 0" fait référence à la variable non déclarée "res".

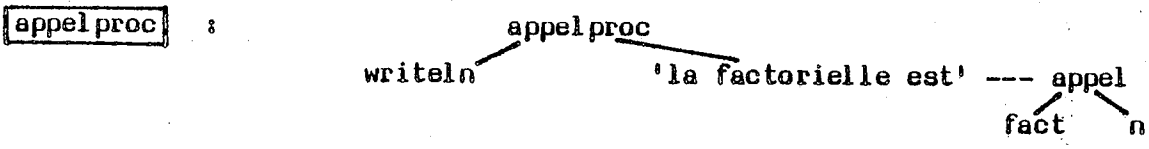
Pour alléger la présentation, nous avons découpé l'arbre en plusieurs morceaux. Un sous-arbre est représenté par un rectangle.











La racine de l'arbre contient un rameau "fourre-tout", permettant de greffer les sous-arbres créés par l'analyse contextuelle. Par exemple, l'identificateur non déclaré "res" y est représenté, afin de toujours avoir un lien entre une utilisation d'identificateur et sa déclaration.

3.7.2. Annotations.

Nous avons défini un certain nombre d'annotations permettant de mémoriser l'analyse contextuelle d'un programme. Les principales annotations sont :

base : repère un arbre descripteur de type :
identificateur prédéfini (integer, real, char, ...),
constructeur de type (intervalle, tableau, ...).

min,max : valeurs entières définies sur un type énumération ou intervalle.

typedecl : type de la déclaration d'un identificateur
(constante, type, variable, paramètre, fonction, indéfini, ...).

declar : lien entre une occurrence d'identificateur et sa déclaration.

autredef : lien entre deux déclarations du même identificateur.

premier, suivant, précédent : liens entre une déclaration d'identificateur et ses occurrences.

table : table des déclarations d'identificateurs dans un bloc lexical
(racine, programme, procédure, fonction, article).

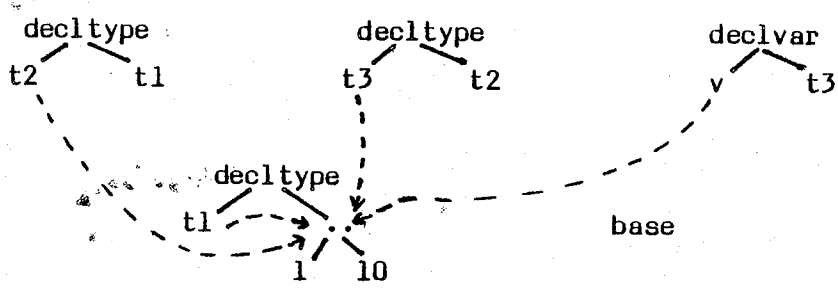
région : bloc de déclaration d'un identificateur.

erreur : code d'erreur (présent sur tous les noeuds).

Nous avons défini les annotations de façon à ne pas avoir de "dictionnaire" séparé de l'arbre ; au contraire, les éléments du dictionnaire sont répartis sur tout l'arbre : ses entrées sont des noeuds "ident", analysés comme des déclarations d'identificateur. Ses valeurs (les types des objets déclarés) sont représentés par des sous-arbres décrivant un type Pascal.

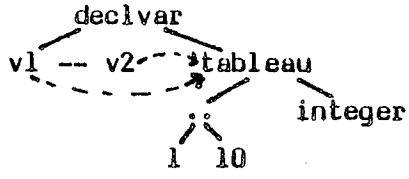
En Pascal, la redéfinition d'un type (ex : type t1 = t2) n'apporte aucune modification sémantique du type. Il est donc possible d'éliminer les "chaînes de référence" sur l'annotation "base".

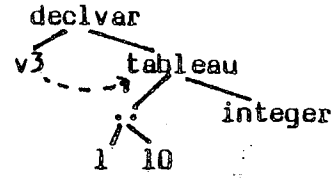
Ex : type t1 = 1 .. 10 ; t2 = t1 ; t3 = t2 ;
var v : t3 ;



Par contre, l'utilisation d'un constructeur de type crée toujours un nouveau type. L'équivalence de types Pascal est réduite à quelques cas simples, comme les entiers et les intervalles d'entiers. Pour nous, deux types sont différents s'ils sont représents par des occurrences d'arbres distinctes.

Ex : var v1, v2 : array [1 .. 10] of integer ;
v3 : array [1 .. 10] of integer ;





L'annotation "declar" est définie sur une occurrence d'identificateur, et possède toujours une valeur (définition correspondante de l'identificateur). La rencontre d'un identificateur non déclaré provoque la création d'un noeud ident, possédant l'annotation erreur = indéfini. C'est le cas de l'instruction "res := 0" de l'exemple.

Enfin, le type retourné par une expression utilisée comme paramètre effectif d'une procédure n'est pas mémorisé. L'annotation "base" décrit dans ce cas le type du paramètre formel correspondant (l'analyse contextuelle vérifie bien sûr la conformité de ces deux types). On trouve une illustration de ce choix dans l'instruction "res := n*fact(n-1)" de l'exemple : "n-1" retourne a priori le type entier ; l'analyse du paramètre formel affecte à l'annotation base le type naturel.

3.7.3. Implémentation.

Nous avons écrit les procédures de manipulation d'arbre de façon à les paramétrer par le langage. La version actuelle d'Adèle ne dispose pas de compilateur de grammaires abstraites ; aussi les tables ont été codées manuellement.

L'implémentation est écrite en Pascal, et simule des enregistrements de taille variable dans un tableau alloué statiquement. Un "enregistrement" représente un noeud de l'arbre et est composé des champs suivants :

oper : nom de l'opérateur (entier),
père : "pointeur" vers l'arbre père,
rang : indice du rameau contenant le noeud,
frère: "pointeur" vers l'arbre suivant dans le rameau,
fils : tableau de n "pointeurs" vers les arbres fils (n est l'arité),
liste: "pointeur" vers la liste des annotations.

Une annotation est représentée par l'enregistrement :
nom : nom de l'annotation (entier),
valeur : valeur de l'annotation (dépend de son type, fourni par une table),
suivant : "pointeur" vers l'annotation suivante.

A la création d'un noeud sont créées les annotations que possèdera toujours le noeud : la donnée d'un opérateur est presque toujours suffisante pour déterminer l'ensemble des annotations que le noeud possède. La seule exception est l'opérateur "ident", dont les annotations dépendent de son contexte d'utilisation : une utilisation d'identificateurs possède les annotations déclar, suivant, précédent, tandis qu'une définition d'identificateur possède les annotations autredef, premier et base. Ces annotations sont créées explicitement par l'analyseur contextuel.

De cette manière, la création d'une annotation est assurée par le noyau de gestion de RI dans la majorité des cas : les outils sont ainsi déchargés de ce problème.

Cette couche "basse" de gestion de RI est interfacée par une couche "haute", assurant principalement cinq fonctions supplémentaires :

- Définition de deux procédures (consultation et modification) pour chaque annotation ; par exemple, l'annotation base est manipulée par cbase et mbase.

- Ajout d'un code d'erreur aux fonctions de manipulation d'arbre (l'emploi incorrect des primitives de la couche basse provoque une erreur d'exécution).

- Simulation des déroutements par appel explicite aux procédures de décompilation, d'analyse contextuelle, etc.

- Prise en compte des noeuds "prédéfinis" : integer, real, etc.

- Archivage d'une RI : la RI est définie de façon à ce que les annotations d'un programme soient des liaisons vers un sous-arbre de ce programme. La seule exception est la liaison vers des noeuds prédéfinis, connus par la couche haute.

L'archivage d'un arbre dans un fichier de caractères consiste en sa linéarisation sous forme préfixée. Les annotations sont mémorisées en même temps que le noeud les portant (seules leurs valeurs sont mémorisées) : la couche "haute" connaît leur ordre de sauvegarde). Les annotations de type "pointeur" sont conservées sous la forme d'un entier, indiquant l'ordre du noeud pointé dans un parcours préfixé de l'arbre (les noeuds prédéfinis ont un ordre constant, quel que soit le programme archivé).

CHAPITRE IV

ANALYSE CONTEXTUELLE INCREMENTALE

Ce chapitre est consacré aux problèmes que pose l'analyse de la syntaxe contextuelle d'un programme lorsqu'elle est intégrée dans un éditeur syntaxique. Après avoir présenté quelques choix possibles et défini la notion d'analyse incrémentale, nous présentons deux approches possibles : l'utilisation de grammaires attribuées classiques et la prise en compte explicite des relations de dépendance entre éléments d'un programme. Nous terminons ce chapitre par l'exposé de la méthode retenue dans Adèle.

Rappelons ce que nous appelons syntaxe hors-contexte, syntaxe contextuelle, et sémantique d'un langage de programmation. La syntaxe hors-contexte (concrète ou abstraite) est l'ensemble des règles de construction d'un programme à l'aide de symboles de base, règles exprimables par une grammaire hors-contexte.

La syntaxe contextuelle est le reliquat des règles de construction que l'on ne peut pas (ou que l'on ne veut pas) exprimer à l'aide d'une grammaire hors-contexte. Cette distinction est justifiée essentiellement par l'usage dans la pratique des grammaires hors-contexte pour écrire des analyseurs syntaxiques.

La syntaxe est constituée de la syntaxe hors-contexte et de la syntaxe contextuelle : elle détermine la conformité des assemblages de symboles, sans leur attribuer une quelconque signification.

La sémantique est le sens donné à un assemblage syntaxiquement valide, c'est-à-dire la fonction que cet assemblage exprime.

4.1. LES CHOIX POSSIBLES.

Nous avons montré qu'un éditeur syntaxique permet à ses utilisateurs de construire des programmes respectant la syntaxe hors-contexte du langage, tandis qu'un interpréteur ou un générateur de code permet d'en tester la sémantique. Qu'en est-il de la syntaxe contextuelle ?

L'utilisateur doit savoir si son programme est correct ou non, et donc une phase d'analyse contextuelle est nécessaire à un moment ou à un autre (de toutes façons, avant l'interprétation). Le problème est donc de déterminer à quel moment effectuer cette analyse :

(a) Après l'édition.

Nous retrouvons le cycle édition-compilation, avec tous ses inconvénients : temps de réponse, manque d'interactivité, liste volumineuse de messages d'erreurs, etc. La situation se dégrade encore plus dans le cas où l'analyseur contextuel est intégré à un compilateur classique : une phase de décompilation, puis d'analyse syntaxique est nécessaire.

Cette solution est la plus simple, car l'écriture d'un analyseur de contexte travaillant à partir d'un arbre est une tâche bien connue, dont la complexité ne dépend que de celle du langage. Pour cette raison, nous qualifions cette solution "d'atelier du pauvre".

Signalons une amélioration de cette approche, permettant une meilleure intégration des outils : l'analyseur du système Légos n'imprime pas de messages d'erreurs, mais il annote les noeuds de l'arbre par un code d'erreur. L'utilisateur peut alors demander à l'éditeur syntaxique de lui montrer les erreurs détectées, les unes après les autres.

(b) Au cours de l'édition.

Cette approche nous semble préférable car elle aboutit à une détection plus précoce des erreurs contextuelles. Le système CPS, par exemple, met en évidence sur l'écran les constructions erronées, lors de chaque modification du programme. L'usager est ainsi immédiatement averti des erreurs qu'il vient de commettre, sans être obligé de les corriger tout de suite (la cause d'une erreur peut être éloignée de son effet). Ce système assure ainsi un grand degré d'interactivité, et constitue un modèle de référence en la matière.

De plus, lorsque l'utilisateur dispose du moyen d'annuler la dernière commande, cette approche présente un réel pouvoir d'apprentissage du langage : l'utilisateur peut modifier un fragment de son programme et en "voir" immédiatement les répercussions. Il peut donc apprendre le langage par essais et erreurs.

Signalons toutefois un inconvénient qui se produit lorsque l'utilisateur effectue une séquence de modifications : les étapes intermédiaires peuvent rendre le programme incorrect, et il s'ensuit la visualisation d'erreurs transitoires qui peuvent distraire l'utilisateur. C'est le cas par exemple lorsqu'une déclaration de variable est modifiée en plusieurs étapes. Mais cette gêne visuelle peut être éliminée si l'utilisateur dispose d'une commande supprimant la mise en évidence des erreurs détectées.

Trois approches sont possibles pour réaliser une analyse contextuelle au cours de l'édition :

- Une passe complète d'analyse du programme sur tout l'arbre après chaque modification ("on efface tout et on recommence"). Cette approche ne peut être viable que sur des programmes de taille réduite, dont le temps de réanalyse est suffisamment court pour rester imperceptible aux utilisateurs. Ce genre de système convient donc bien pour des étudiants en informatique, mais est impraticable pour des applications professionnelles.

- Une passe complète d'analyse sur un sous-arbre (par exemple une procédure) lorsque l'utilisateur "déclare" avoir fini de le modifier. Par exemple, dans Gandalf, une procédure est complètement réanalysée dès lors que l'utilisateur positionne le curseur ailleurs. Ce mécanisme assure des temps de réponse convenables, mais l'utilisateur n'est pas averti des erreurs au moment où il modifie un fragment de la procédure. On retrouve donc, sous une forme atténuée, le manque d'interactivité signalé dans le cycle édition-compilation. Mais l'écriture de l'analyseur contextuel reste simple : très peu d'aménagements techniques sont à prévoir par rapport à un analyseur global.

- Une passe d'analyse uniquement sur le sous-arbre modifié (et sur les sous-arbres contextuellement corrélés). Cette approche est celle qui assure le plus grand degré d'interactivité avec le meilleurs temps de réponse : c'est "l'atelier du riche". C'est également celle qui pose le plus de problèmes de réalisation. La suite de ce chapitre est consacrée à cette approche.

4.2. PRESENTATION DU PROBLEME.

Il est possible de modéliser le résultat d'un analyseur contextuel comme un ensemble d'annotations décorant l'arbre abstrait. Par exemple, le processus d'identification des identificateurs peut être perçu comme l'établissement d'un lien de chaînage entre deux noeuds "ident" (la source de ce lien est une occurrence de l'identificateur, et la cible en est la déclaration). La vérification de type aboutit à la mémorisation du type retourné par une expression, des opérations de conversion de type à effectuer, ..., ou du code de l'erreur détectée.

Un analyseur contextuel consiste alors en un parcours judicieux de l'arbre, permettant l'évaluation de l'ensemble résultant d'annotations. Un ou plusieurs passages sur l'arbre peuvent être nécessaires, suivant la complexité du langage. En pratique, un parcours préfixé de l'arbre est suffisant pour un langage assez simple comme Pascal.

Un analyseur contextuel couplé à un éditeur est activé à chaque opération modifiant l'arbre : insertion, suppression, remplacement d'un sous-arbre, etc. Toutes les annotations possèdent une valeur avant et après la modification ; seul un sous-ensemble des annotations change de valeurs, mais l'analyseur (qui ne connaît pas a priori ce sous-ensemble) réévalue en fait un plus grand sous-ensemble d'annotations, parfois l'ensemble tout entier.

Un analyseur contextuel est dit incrémental si l'ensemble des annotations réévaluées est différent de l'ensemble total des annotations. Il est dit optimal si son coût est proportionnel au nombre d'annotations modifiées.

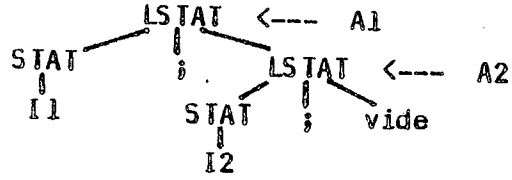
La notion d'optimum est largement tributaire de la définition des annotations : il est possible de définir deux ensembles d'annotations, globalement équivalentes, mais dont le nombre total varie extrêmement. Pour chaque définition, il est possible de développer un analyseur qualifié d'optimal ; mais le temps de réponse varie extrêmement. Par exemple, les annotations issues d'une grammaire attribuée sont en grand nombre, car ce formalisme impose des relations de proche en proche dans l'arbre. Au contraire, si les relations de dépendance lointaine entre sous-arbres sont matérialisées par des annotations, le nombre total d'annotations est très nettement inférieur. Mais, malgré l'insuffisance de cette notion d'optimum, nous nous en contenterons, en l'absence d'un meilleur critère.

En résumé, l'écriture d'un analyseur incrémental pose donc le problème de minimiser le nombre d'annotations à réévaluer, lors de la modification d'un arbre (le déclenchement de l'analyse est facilement obtenu par une procédure de déroutement). Deux approches semblent possibles : l'utilisation de grammaires attribuées, comme c'est le cas dans CPS <Reps 82,83>, et la prise en compte des relations de dépendance lointaine <Fischer 82>.

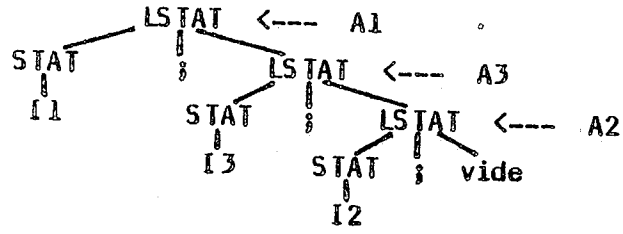
Ces deux méthodes sont développées sur l'arbre de la syntaxe concrète du programme. La forme d'un arbre concret simplifie en effet le modèle d'édition, qui se réduit à la création et au remplacement de sous-arbres. Considérons par exemple l'insertion d'une instruction dans une liste d'instructions : cette notion de liste est traduite par un arbre concret "binaire", et l'insertion se traduit par un remplacement.

Ex : grammaire : LSTAT ::= ε , LSTAT ::= STAT ; LSTAT

La "liste" : "I1, I2" est représentée par l'arbre :



L'insertion de I3 entre I1 et I2 produit l'arbre :



L'insertion se décompose en une création et trois remplacements d'arbres :

- création de A3 :

- remplacement dans A1 de A2 par vide ;
- remplacement dans A3 de vide par A2 ;
- remplacement dans A1 de vide par A3.

Pour respecter la méthode des auteurs, nous présentons leurs approches à l'aide d'arbres concrets et de grammaires concrètes. Nous supposerons que tous les symboles non-terminaux d'une grammaire possèdent une règle "X ::= vide", permettant la prise en compte de programmes incomplets.

4.3. UTILISATION DE GRAMMAIRES ATTRIBUEES.

4.3.1. Grammaires attribuées.

Le formalisme des grammaires attribuées a été défini par Donald Knuth <Knuth 68> pour la description de la "signification" des programmes. On trouvera en <Lorho 74> une étude complète de ce formalisme. Nous nous limitons ici à un exemple rappelant les notions qui nous sont utiles.

Les attributs sont des valeurs attachées aux symboles d'une grammaire hors-contexte. Ils sont définis par des fonctions d'évaluation associées aux règles dans lesquelles ils apparaissent.

Un attribut est dit synthétisé s'il est associé au non-terminal partie gauche d'une règle, et hérité s'il est associé à un symbole contenu dans une partie droite de règle. Sur l'arbre concret, le calcul d'un attribut synthétisé permet de "remonter" de l'information du sous-arbre correspondant à l'expansion d'un non-terminal, tandis qu'un attribut hérité permet de "redescendre" une information dans un sous-arbre.

On impose de plus la non-circularité dans le graphe de dépendance des attributs à évaluer, sur tout arbre représentant un programme syntaxiquement correct.

On dit qu'un attribut a1 est le successeur de a2 si l'évaluation de a1 dépend de la valeur de a2. Une grammaire attribuée non circulaire est donc telle qu'aucun attribut n'est transitivement son propre successeur.

Ex : soit la grammaire "définissant" des arbres binaires (a, aba, ababa, ...) :

Vt = {a,b}

Vn = {S,R} (axiome S)

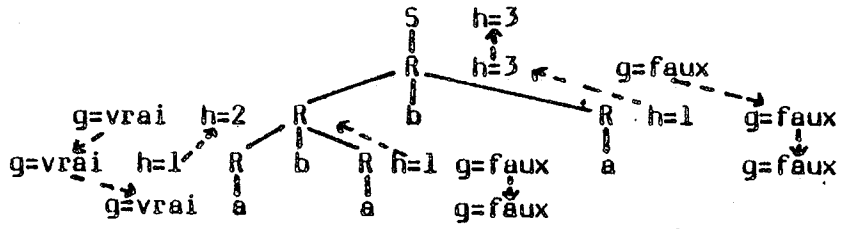
P = {S ::= R, R ::= R b R, R ::= a}.

Nous définissons un attribut synthétisé, h (hauteur de l'arbre) et un attribut hérité, g (feuille gauche) dont les fonctions d'évaluation sont :

S ::= R	h(S) = h(R)	g(R) = faux	
R ::= R b R	h(R0)= 1+ max(h(R1),h(R2))	g(R1)= vrai	g(R2)= faux
R ::= a	h(R) = 1	g(a) = g(R)	

(Lorsqu'un même symbole possède plusieurs occurrences dans une règle, elles sont distinguées dans les fonctions d'évaluation en les numérotant par leur ordre d'apparition dans la règle, de gauche à droite).

Soit à analyser la phrase "a b a b a". L'arbre décoré est :



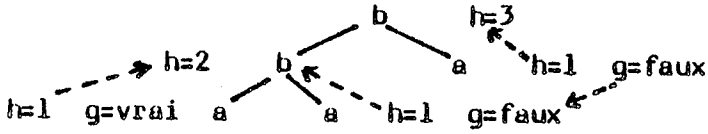
Beaucoup de règles d'évaluation d'attributs sont simplement des copies de valeurs : ceci traduit le fait que les attributs doivent se propager "de proche en proche".

L'expérience montre que l'écriture et la compréhension d'une grammaire attribuée sont rendues compliquées par le nombre des attributs et des fonctions d'évaluation. Nous pensons que l'utilisation de ce formalisme serait simplifiée s'il était défini sur une grammaire abstraite. Notre exemple deviendrait :

S ::= a, b

b --> S, S h(b) = 1 + max(h(S1), h(S2)) g(S1) = vrai g(S2) = faux

a --> h(a) = 1



4.3.2. Analyse par propagation des modifications.

Un arbre est dit complètement attribué si tous ses attributs ont une valeur. Le plus souvent, les attributs hérités de la racine de l'arbre ne peuvent pas être évalués si la racine ne correspond pas à l'axiome de la grammaire, c'est-à-dire si l'arbre peut correspondre à un sous-arbre d'un programme syntaxiquement correct. Dans ce cas, les attributs hérités reçoivent une valeur arbitraire.

Un attribut est disponible s'il possède une valeur. Un attribut est cohérent si ses prédécesseurs sont disponibles, et si sa valeur est bien celle que retourne sa fonction d'évaluation.

Soit A1 et A2 deux arbres complètement attribués et A3 un sous-arbre de A1. Le remplacement de A3 par A2 provoque l'analyse suivante :

affecter aux attributs synthétisés de la racine de A3 les valeurs des attributs correspondants de A2
S := {attributs de la racine de A3 devenus incohérents}
tantque $S \neq \emptyset$ faire
    choisir un élément a de S
    S := S - {a}
    a^0 := a
    évaluer a
    si $a \neq a^0$ alors S := S ∪ {successeurs de a}
fin

Pendant le processus d'évaluation, tous les attributs sont disponibles, mais peuvent être incohérents. Le fait d'affecter une nouvelle valeur à un attribut rend incohérents tous ses successeurs, ce qui nécessite leur réévaluation. Cette propagation s'arrête lorsque la nouvelle valeur d'un attribut est égale à son ancienne.

L'initialisation de l'algorithme consiste à déterminer le sous-ensemble des attributs hérités de la racine de A2 qui sont incohérents. Rappelons que ces attributs ont reçu une valeur arbitraire, afin de rendre A2 complètement attribué.

Cet algorithme n'est certainement pas optimal car son coût dépend de l'ordre dans lequel sont choisis les attributs à réévaluer. Supposons par exemple qu'un attribut a dépende de b et c, et que S contienne a, b et c : si l'on choisit d'évaluer a, puis b, puis c, a sera évalué trois fois.

<Reps 83> donne un exemple où le choix d'évaluation "premier entré, premier sorti" entraîne un coût quadratique de l'algorithme, et le choix "dernier entré, premier sorti" donne un coût exponentiel.

4.3.3. Analyse par annulation-réévaluation.

Un attribut est dit prêt à être évalué si ses prédécesseurs sont tous disponibles.

L'algorithme suivant consiste à annuler (c'est-à-dire rendre indisponibles) tous les attributs concernés par un remplacement d'arbre, puis à les réévaluer.

Soit A1 et A2 deux arbres complètement attribués, et A3 un sous-arbre de A2. Le remplacement de A3 par A2 provoque l'analyse :

```
affecter aux attributs synthétisés de la racine de A3 les attributs
    correspondants de A2
(* annulation *)
S := {attributs de la racine de A3}
tantque S ≠ ∅ faire
        choisir un attribut a de S
        annuler la valeur de a
        S = S - {a} U {successeurs disponibles de a}
fin
(* réévaluation *)
S :=  attributs de la racine de A3 prêts à être évalués
tantque S ≠ ∅ faire
        choisir un élément a de S
        S := S - {a}
        évaluer a
        S := S U {successeurs de a prêts à être évalués}
fin
```

Cet algorithme est plus "efficace" que le précédent, puisqu'il garantit que l'évaluation d'un attribut n'est effectuée qu'une seule fois. Il n'est cependant pas optimal, car il peut évaluer des attributs dont la valeur ne change pas.

4.3.4. Analyse "optimale" par propagation des modifications.

La critique que nous avons faite de la version initiale de cet algorithme donne une idée de son amélioration : il faut trouver un ordre pour choisir les éléments de l'ensemble S, ordre assurant que l'évaluation d'un attribut lui affecte directement sa valeur finale, ce qui entraîne que cette évaluation n'a lieu qu'une seule fois.

Reps décrit un algorithme généralisant le tri topologique de Knuth. Le lecteur trouvera des explications détaillées dans <Reps 83> : nous ne présentons que les grands principes de cet algorithme. Si a et b sont deux attributs portés par un même noeud, et si b dépend transitivement de a, alors a doit être évalué avant b. La principale différence avec l'algorithme classique de tri topologique est que le graphe de dépendance est construit dynamiquement, au fur et à mesure de l'évolution de l'ensemble des attributs à réévaluer.

Soit A1, A2 deux arbres complètement attribués, et A3 un sous-arbre de A1. Le remplacement de A3 par A2 provoque l'analyse :

```
affecter aux attributs synthétisés de la racine de A3 les valeurs des
     attributs correspondants de A2.
G := graphe de dépendance des attributs de la racine de A3
S := {sommets de G sans prédécesseurs dans G}
T := S
tantque S ≠ ∅ faire
     choisir un élément a de S
     S := S - {a}
     a⁰ := a
     évaluer a
     si (a⁰ = a) et (G ne contient pas tous les successeurs de a)
     alors
          G := G U {graphe des successeurs de a}
          T := T U {successeurs de a}
     fin
     pour tous les successeurs b de a faire
          enlever l'arc a --> b dans G
          si (b ∈ T) et (b n'a pas de successeur dans G)
          alors S := S U {b} fin
     fin
fin
```

T contient les attributs susceptibles d'être réévalués, et S les attributs à réévaluer. A la fin de l'algorithme, G contient tous les attributs dépendant d'un attribut dont la valeur a changé. Les seuls attributs effectivement réévalués sont les sommets isolés de G.

Le coût de l'expansion de G est borné par la taille de la plus grande règle de la grammaire, puisque les dépendances entre attributs sont locales. D'autre part, les seuls éléments ajoutés à S sont les attributs modifiés, et ils n'y sont rajoutés qu'une seule fois. Le coût global de l'algorithme est donc proportionnel au nombre d'attributs dont la valeur a été modifiée : l'algorithme est "optimal".

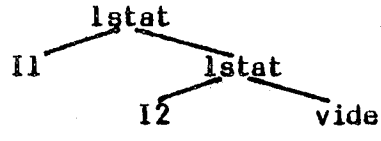
4.3.5. Critique de la méthode.

Nous avons signalé au paragraphe 4.2 que l'utilisation de l'arbre concret simplifie le modèle d'édition, puisque toute opération peut se traduire en une séquence de remplacements d'arbres.

Mais cette simplification a pour conséquence qu'une opération d'édition telle que l'insertion d'une instruction dans une liste provoque plusieurs appels à l'analyseur incrémental. Chaque réanalyse peut être qualifiée d'"optimale", mais l'effet global manque d'efficacité, puisqu'un même attribut peut être réévalué plusieurs fois.

<Reps 83> propose une modification des algorithmes d'analyse permettant une seule réanalyse pour un ensemble d'opérations de remplacement d'arbres. Malheureusement, cette amélioration n'est pas possible pour l'insertion d'un élément dans une liste, car les remplacements d'arbres ne sont pas indépendants.

La méthode d'analyse de Reps peut être adaptée aux arbres abstraits, pourvu qu'ils ne possèdent pas la notion de liste. Par exemple, une liste "I1 ; I2" doit être représentée par l'arbre :



Mais l'insertion dans une liste reste toujours inefficace.

Nous pensons qu'il est possible d'adapter le formalisme des grammaires attribuées, et par suite, l'analyse incrémentale, aux arbres abstraits tels que

nous les avons définis. Le problème principal à résoudre est l'expression de la dépendance des attributs dans une liste d'arbres. A notre connaissance, ce problème reste ouvert.

Les attributs deviennent évidemment des annotations sur les noeuds de l'arbre abstrait, ce qui résoud le problème de leur mémorisation. De plus, l'arbre abstrait étant plus concis que l'arbre concret, le nombre d'attributs réévalués est moins important (la majorité des fonctions d'évaluation sur l'arbre concret sont des règles de recopie), et l'analyse est plus efficace.

Mais la principale limitation de l'approche de Reps est due au formalisme des grammaires attribuées, qui impose des relations locales de dépendance d'un attribut : l'évaluation de l'attribut d'un noeud de l'arbre concret ne peut dépendre que des attributs portés par ce noeud, ses frères et son père.

Il n'est donc pas possible d'exprimer simplement la relation existant entre une occurrence d'identificateur et sa déclaration. On utilise généralement un attribut "table d'identificateurs" dont la valeur est modifiée sur les noeuds de déclaration d'identificateur, et qui est transmis de proche en proche dans tout l'arbre, jusqu'aux noeuds occurrence d'identificateur.

L'analyse incrémentale effectuée lors de la mise à jour d'une déclaration d'identificateur doit donc parcourir la presque totalité de l'arbre, pour la mise à jour de tous les attributs "table d'identificateurs". L'algorithme reste "optimal" au sens où nous l'avons défini, mais l'analyse serait plus efficace si elle pouvait déterminer directement les noeuds concernés par la mise à jour de la déclaration. Dans ce cas, il serait inutile de définir l'attribut "table d'identificateurs" sur tous les noeuds de l'arbre, et donc le nombre total de noeuds visités et d'attributs réévalués serait très nettement moins important.

La deuxième approche proposée par Johnson et Fisher <Johnson 82> propose justement un modèle de grammaire attribuée tenant compte de ces relations lointaines entre noeuds de l'arbre. Mais nous montrerons que cette approche est hélas actuellement limitée.

4.4. RELATIONS DE DEPENDANCE LOINTAINE.

4.4.1. Ensembles de relations contextuelles.

Soit G une grammaire attribuée. Un ensemble de relations contextuelles (ERC) est la donnée de :

- un sous-ensemble ordonné de symboles de G,
- un ensemble de fonctions d'évaluation d'attributs,
- une assertion, c'est-à-dire un prédicat logique sur les attributs d'un arbre syntaxique.

Une occurrence d'ERC est un ensemble de noeuds de l'arbre syntaxique, étiquetés par un élément du sous-ensemble de symboles de l'ERC, et satisfaisant l'assertion de l'ERC.

Exemple : ensemble $\{X,Y\}$
fonction compte (X) := compte (X) + 1
assertion le noeud Y est un descendant du noeud X

L'attribut compte (X) désigne le nombre total de descendants Y de X. L'association d'un noeud X avec tous ses descendants Y aboutit à la valeur finale de l'attribut compte (X).

Une différence fondamentale entre ce formalisme et une grammaire attribuée, est que les attributs ne respectent pas la règle d'assignation unique. En pratique, la rencontre d'un symbole X permet d'initialiser ses attributs (comme dans une grammaire attribuée), tandis que la détection d'une association de X dans un ERC entraîne la mise à jour de ses attributs.

Exemple : la grammaire suivante définit la syntaxe contextuelle d'un langage à structure de blocs, permettant l'affectation de variables de même type, entier ou caractère.

```
BLOC  ::= begin LDECL LSTAT end
LDECL ::= ε
LDECL ::= DECL LDECL
DECL  ::= id : TYPE        type(id) := type(TYPE)   occ(id) := def
TYPE  ::= integer          type(TYPE) := entier
TYPE  ::= char             type(TYPE) := caract
LSTAT ::= ε
```

LSTAT ::= STAT LSTAT

STAT ::= BLOC

STAT ::= id := id type(STAT) := si type(id1) = type(id2) alors type(id1) sinon nil

occ(id1) := util occ(id2) := util

Cette grammaire définit deux attributs : type (entier ou caractère) et occ (nature de l'identificateur : définition ou utilisation).

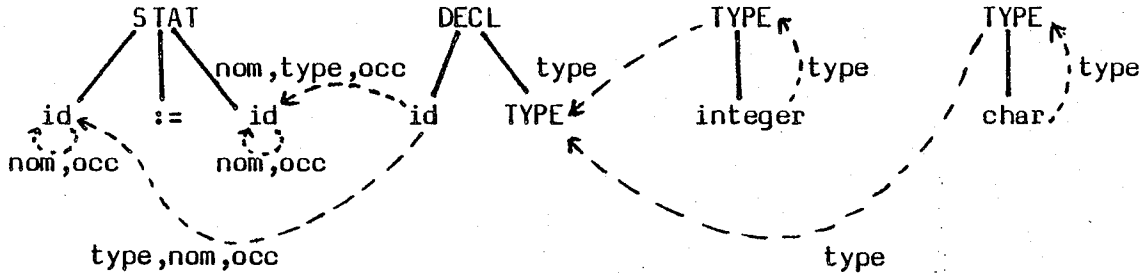
ERC identificateurs :

ensemble {id}

association (id1, id2) si : (nom(id1) = nom(id2))
et (occ(id1) = def) et (occ(id2) = util)
et (tout arbre BLOC contenant id1 contient id2)
et (tout arbre BLOC contenant id2 ne contient pas de id3 = id1 tel que nom(id3) = nom(id1))

fonction : type(id2) := type(id).

Le graphe de dépendance des attributs est le suivant :



Le type d'une utilisation d'identificateur dépend du type de sa déclaration par la définition de la fonction d'évaluation de l'ERC, mais il dépend aussi du nom et du type d'occurrence de lui-même et de sa déclaration par la définition de l'assertion de l'ERC. On peut constater que le graphe de dépendance des attributs ne contient pas de circuits, contrainte que nous supposerons dans toute la suite.

<Johnson 82> impose de plus que l'assertion d'un ERC ne dépende que d'attributs de "degré nul", c'est-à-dire dont la fonction d'évaluation ne dépend que d'attributs possédant une valeur immédiate : c'est le cas des attributs "nom" et "occ" dans notre exemple. Cette restriction sévère est justifiée par le fait que l'évaluation est simplifiée et que le formalisme reste suffisamment puissant.

4.4.2. Evaluation non incrémentale.

L'évaluation des attributs suppose l'existence de l'arbre concret dans sa totalité. Elle fait appel à un ensemble des "associations à évaluer" : une association est soit une occurrence d'ERC (avec ses fonctions d'évaluation), soit une occurrence de règle de la grammaire (avec ses fonctions d'évaluation).

Un noeud de l'arbre concret peut participer à au plus deux associations de type règle : son étiquette peut être partie gauche d'une règle ou élément d'une partie droite de règle. Un noeud de l'arbre peut de plus participer à un nombre quelconque d'associations de type ERC.

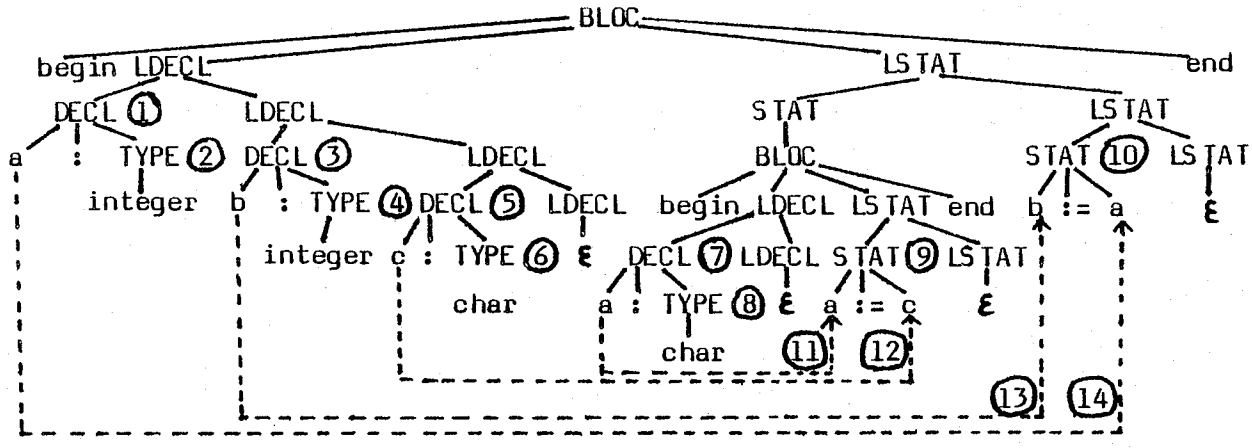
Une association est dite prête à être évaluée si tous les attributs dont elle dépend (par le biais des fonctions d'évaluation) sont disponibles.

L'évaluation est la suivante :

```
S := ∅
pour  tout attribut a de degré nul faire
      évaluer a
      S := S ∪ {associations dépendant de a prêtes à être évaluées}
fin
pour  toute occurrence d'ERC vérifiant son assertion faire
      créer l'occurrence d'ERC
      si l'association A correspondant à l'ERC est prête à être évaluée
      alors  S := S ∪ {A}    fin
fin
tantque  S ≠ ∅  faire
      choisir une association A de S
      S := S - {A}
      évaluer les fonctions de A
      pour tous les attributs a ayant changé de valeur faire
          S := S ∪ {associations dépendant de a prêtes à être évaluées}
      fin
fin
```

Nous allons illustrer le mécanisme d'évaluation sur le programme : "begin a : integer b : integer c : char begin a : char a := c end a := b end", correspondant à la grammaire du paragraphe 4.4.1.

L'analyse syntaxique produit l'arbre suivant :



Nous avons numéroté de 1 à 10 les associations de type règle, et de 11 à 14 les associations de type ERC.

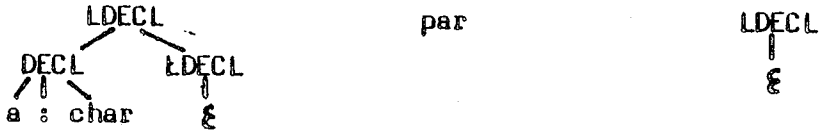
Les associations 2, 4, 6, 8 ne dépendent que d'attributs de degrés nuls, et sont donc évaluées dans la première phase de l'algorithme. Les associations 1, 3, 5, 7 ne dépendent que de l'attribut type qui vient d'être évalué, et forment donc l'ensemble initial des associations prêtes à être évaluées. La deuxième phase de l'algorithme produit les occurrences d'ERC de 11 à 14, qui ne sont pas encore prêtes à être évaluées. La troisième phase de l'algorithme est décrite par le tableau suivant :

associations prêtes	choix	attributs évalués	nouvelles assertions
1, 3, 5, 7	1	type(a) = integer	14
3, 5, 7, 14	3	type(b) = integer	13
5, 7, 14, 13	5	type(c) = char	12
7, 14, 13, 12	7	type(a) = char	11
14, 13, 12, 11	14	type(a) = integer	∅
13, 12, 11	13	type(b) = integer	10
12, 11, 10	12	type(c) = char	∅
11, 10	11	type(a) = char	9
10, 9	10	type(STAT) = integer	∅
9	9	type(STAT) = char	∅
∅			

4.4.3. Evaluation incrémentale.

L'algorithme d'évaluation utilisé lors du remplacement d'un sous-arbre fait appel à trois phases, dont la dernière est du type propagation des modifications (voir le paragraphe 4.3.2.).

Pour illustrer l'algorithme, nous reprenons l'exemple précédent, dans lequel nous supprimons la déclaration "a : char".
Cette destruction se modélise par le remplacement de :



1ère phase : destruction des occurrences d'ERC devenues invalides. Ce cas se produit quand les noeuds participant à une occurrence d'ERC ne sont pas tous "à l'intérieur" du sous-arbre à remplacer, ou quand ils ne sont pas tous "à l'extérieur".

Exemple : le remplacement de "a : char" détruit l'occurrence d'ERC 11.

La destruction d'une occurrence d'ERC provoque l'appel d'une procédure d'invalidation, écrite à la main (les auteurs ne fournissent aucune indication sur la façon de la produire automatiquement : l'invalidation doit être résolue cas par cas).

Par exemple, le remplacement d'une déclaration d'identificateur provoque le traitement suivant :

- chercher une déclaration du même identificateur respectant les règles de visibilité,
- établir une occurrence d'ERC entre la déclaration trouvée et les utilisations d'identificateurs participant à l'occurrence d'ERC qui vient d'être détruite.

Cette première phase permet de réunir un ensemble initial d'attributs devant être réévalués.

Exemple (suite) : invalidation de l'ERC 11 reliant "a : char" à "a := c". Cette opération aboutit à la création de l'ERC 15, entre "a : integer" et "a := c". L'association 15 est prête à être évaluée.

2ème phase : création des occurrences d'ERC produites par l'insertion du nouveau sous-arbre. Là encore, ce traitement fait appel à des procédures écrites à la main.

Par exemple, l'insertion d'une déclaration d'identificateur provoque le traitement suivant :

```
pour tous les blocs contenant la déclaration faire
    si la déclaration en "masque" une autre
    alors mettre à jour l'occurrence d'ERC de la déclaration masquée.
```

Exemple (suite) : la dernière phase n'a aucun effet car l'insertion du nouveau sous-arbre ne crée pas d'ERC.

3ème phase : les deux premières phases ont déterminé l'ensemble S initial des associations prêtes à être évaluées. La troisième phase est la propagation des modifications :

```
tantque S ≠ ∅ faire
    choisir et évaluer une association A
    S := S - {A}
    pour tous les attributs a ayant changé de valeur faire
        S := S U {associations dépendant de a}
    fin
fin
```

Exemple : S ne contient au départ que l'association 15 (créée à la première phase). L'évaluation de 15 aboutit à "type(a) = integer" dans "a := c". La valeur de type(a) ayant changé, l'association 9 devient prête à être évaluée (et c'est la seule). Son évaluation aboutit à "type(STAT) = nil", puis l'algorithme s'arrête.

4.4.4. Critique de la méthode.

Le grand avantage de cette méthode est de permettre à l'analyseur incrémental de connaître rapidement quels sont les attributs concernés par une modification. Dans l'exemple que nous avons développé, la première phase de l'analyse détermine tout de suite la nécessité de réévaluer l'instruction "a := c", alors que cette instruction peut être arbitrairement éloignée de la déclaration supprimée "a : char".

La méthode de Reps aurait abouti à propager la nouvelle table de symboles dans toutes les instructions du bloc intérieur, quel que soit leur nombre. On conçoit donc facilement l'efficacité de cette méthode d'analyse contextuelle.

Un premier inconvénient est que l'algorithme proposé n'est pas "optimal", puisqu'il correspond à une simple propagation des modifications. Nous pensons cependant qu'une étude plus approfondie permettrait de le rendre "optimal".

Un problème de conception se pose lors de la définition des ERC : quelles sont les dépendances lointaines qu'il faut mémoriser sous forme d'ERC ? Il semble que l'efficacité de l'algorithme dépende étroitement de la définition des ERC.

Il ne faut pas, de plus, définir "trop" d'ERC : l'algorithme perdrait finalement plus de temps à mettre à jour les relations qu'à les exploiter. Un compromis judicieux est donc à trouver, mais il ne s'est dégagé aucune méthodologie pour aider le concepteur dans ce choix. Tout au plus pensons-nous que les ERC traitant des identificateurs sont indispensables.

Le plus grave défaut de cette méthode est finalement l'impossibilité de produire automatiquement un analyseur contextuel utilisant les ERC : leur mise à jour doit être écrite à la main, et nous voyons mal comment déduire automatiquement ces actions à partir d'une grammaire attribuée.

Néanmoins, les grands principes de cette méthode peuvent servir de guide à l'écriture manuelle d'un analyseur contextuel incrémental. C'est cette démarche que nous avons expérimentée dans l'atelier Adèle.

4.5. ANALYSE INCREMENTALE DANS L'ATELIER ADELE.

4.5.1. Principe.

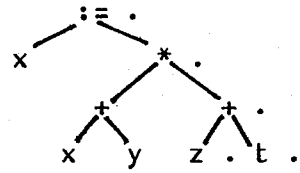
Le principe retenu est un algorithme d'annulation-réévaluation, tenant compte des relations de dépendance lointaine. On peut donc dire immédiatement que l'algorithme n'est pas "optimal". Ce choix a été pris à cause de sa simplicité d'implémentation dans un algorithme classique d'analyse effectuant un parcours récursif descendant de l'arbre.

Nous utilisons la notion d'incrément d'analyse : un incrément est le plus petit sous-arbre concerné par une modification, quand on ne tient compte que des relations de dépendance rapprochée. Les déclarations et les instructions Pascal sont des incréments. L'annulation des annotations ne remonte jamais au-delà d'un noeud incrément, et consiste en un marquage des noeuds concernés.

Nous ne mémorisons que deux types de dépendance lointaine : l'ensemble des définitions d'un identificateur et l'ensemble des utilisations associées à une définition.

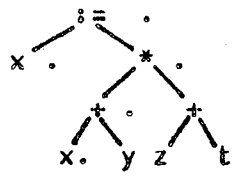
L'analyse d'une modification consiste à déterminer (via les dépendances lointaines) l'ensemble des incréments concernés par les modifications, puis à les analyser. L'analyse d'un incrément peut aboutir à de nouvelles dépendances lointaines, et donc augmenter l'ensemble des incréments à réanalyser. L'analyse consiste en une descente récursive, utilisant des paramètres valeurs (attributs hérités) et résultats (attributs synthétisés). La descente peut être tronquée à la rencontre d'un noeud non annulé : il y a alors consultation des annotations associées à ce noeud, pour la transmission des attributs synthétisés.

Exemple 1 : x := (x+y) * z est modifié en x := (x+y) * (z+t).



Les noeuds annulés sont ceux qui sont marqués d'un point. L'analyse descendante récupère les annotations de x et  , qui ne sont pas marqués.

Exemple 2 : la déclaration de x est modifiée. Les relations lointaines déterminent la réanalyse de "x := (x+y) * (z+t)". Les noeuds marqués sont alors :



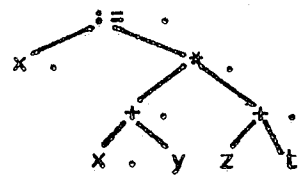
Les arbres y et  ne sont pas réanalysés.

Remarque : plusieurs ensembles de relations lointaines peuvent déterminer la réanalyse du même incrément. Dans la quasi-totalité des cas, cet incrément ne sera réanalysé qu'une seule fois.

Ex : type u = ...
var x : u
var z : u
x := (x+y) * (z+t)

Examinons le cas d'une modification du type de t. L'ensemble E des incréments à réanalyser vaut initialement : E = {type t = ...}
L'analyse de type "t=..." modifie E, qui devient : E = {var x : t, var z : t}
Après analyse de "var x:t" : E = {var z : t, x := (x+y) * (z+t)}
Enfin, après analyse de "var z:t" : E = {x := (x+y) * (z+t), x := (x+y) * (z+t)}

Les noeuds marqués sont :



Cette solution aboutit à une seule réanalyse de cet arbre, car en Pascal les déclarations précèdent les utilisations d'identificateurs. La gestion de E est alors très simple : c'est une liste de type "premier entré, premier sorti". L'arbre "x := (x+y) * (z+t)", bien que mémorisé deux fois dans E, n'est analysé qu'une seule fois, car cette analyse ôte les marques des noeuds de l'arbre. A la rencontre de la deuxième occurrence de cet arbre dans E, l'analyse n'est pas effectuée car la racine de l'incrément n'est plus marquée.

La seule exception à ce mécanisme est la définition d'un type pointeur, pouvant utiliser un identificateur de type non encore déclaré. L'incrément "type pointeur" est alors analysé deux fois.

4.5.2. Algorithmes.

Les algorithmes utilisés sont décrits en pseudo-Pascal. Les annotations sont manipulées par deux primitives : la consultation (le nom de l'annotation est par "c", ex. cbase) et la modification (le préfixe est "m").

```
remplacer (ancien, nouveau)
(* cette procédure est appelée après le remplacement de ancien par nouveau *)
    E := ∅
    annuler (ancien)
    analyse (incrément (nouveau), région (nouveau))
    tantque  E ≠ ∅  faire
      retirer le premier élément a de E
      si a est marqué alors analyse (a, région (a))
    fin

annuler (a)
(* cette procédure marque tous les noeuds de a et détermine les incréments
   dépendant d'une relation lointaine *)
    pour tous les noeuds b de a faire
      marquer b
      si b est une déclaration d'identificateur alors
         sortir b de la table d'identificateurs
         sortir b de l'ensemble des redéfinitions
         si (∃ une définition c du même identificateur dans le même bloc)
         et (cette définition n'est pas contenue dans a)
         alors  E := E U {incrément (c)}
         pour toute utilisation c de l'identificateur b faire
            si c n'est pas contenue dans a alors  E := E U {incrément (c)}
         fin
      fin
    fin

incrément (a)
    marquer a
    tantque  a n'est pas un incrément faire
         a := père (a)
         marquer a
    fin
    retourner a
```

```
région (a)
(* retourne le plus proche ancêtre de a définissant une région de définition
   des identificateurs : racine, prog, declproc, declfonct, article *)
    tantque a n'est pas une région faire a := père (a)
    retourner a

analyse (a, r)
(* a est le sous-arbre à analyser, r est la région de a *)
    si a est marqué alors
       démarquer a
       erreur := correct
       choix oper (a) dans
         .....
         decltype :  descripteur (fils (a,2), r, base2, erreur2)
                     déclaration (fils (a,1), r, base2, erreur2, erreur)
                     (* base2 et erreur2 sont synthétisés par descripteur, et
                        hérités par déclaration, qui synthétise erreur *)
       affectation:  référence (fils (a,1), r, base1, erreur1)
                     expression (fils (a,2), r, base2, erreur2)
                     si erreur1 ou erreur2 n'est pas correct
                     alors erreur := erreur-interne
                     sinon  compatibilité (base1, base2, base, erreur)
                            (* retourne le type de base compatible avec base1
                               et base2 et renvoie un code d'erreur *)
                            mbase (a, base)
                     fin
         si-alors :  expression (fils(a, 1), r, base1, erreur1) (* condition *)
                     si erreur1 = correct alors erreur := erreur-interne
                     sinon si base1 = prédéfini ("boolean")
                           alors erreur := erreur-logique
                     analyse (fils (a, 2), r)     (* partie alors *)
                     analyse (fils (a, 3), r)     (* partie sinon *)
         .....
       fin choix
       merreur (a, erreur)          (* mise à jour de l'annotation erreur *)
    fin
```

Nous avons montré sur trois exemples l'analyse dirigée par la syntaxe. A part le prélude des procédures et la mise à jour des annotations, l'écriture est identique à celle que l'on trouverait dans un analyseur non incrémental. La complexité d'écriture de ces procédures se réduit donc à celle du langage Pascal. Les seules procédures complexes sont l'analyse d'une déclaration et d'une utilisation d'identificateur, car il faut mettre à jour les relations de dépendance lointaine.

```
utilisation (a, r, base, erreur)
(* cette procédure est appelée à la rencontre d'une utilisation d'identifica-
   teur a dans la région r ; elle synthétise le type de base de a et un code
   d'erreur *)
    si a n'est pas marqué alors
       base := cbase (cdeclar (a))
       erreur := cerreur (a)
    sinon
       démarquer a
       trouver une déclaration b de l'identificateur dans une région visible
       de r (* s'il n'y en a pas, il y a création de b, portant l'annotation
               erreur = indéfini *)
       ajouter a à l'ensemble des utilisations de b
       mdeclar (a, b)
       merreur (a, cerreur (b))
       erreur := cerreur (b)
       base := cbase (b)
    fin

déclaration (a, r, base, erreurl, erreur)
(* cette procédure est appelée à la rencontre d'une déclaration d'identifica-
   teur a dans la région r ; le type de a est base, et peut être erroné
   (erreurl) ; erreur est un code de retour *)
    si a n'est pas marqué alors
       si (base ≠ cbase (a)  ou (erreurl ≠ cerreur (a)) alors
          mbase (a, base)
          merreur (a, erreurl)
          pour tout identificateur b relié à a faire  E := E ∪ {incrément (b)}
          si erreur l ≠ correct alors erreur := erreur-interne
                                sinon erreur := correct
       fin
    sinon
```

```
        si il existe d'autres déclarations de l'identificateur dans r alors
            erreur := double-définition
            si il existe une seule déclaration b de l'identificateur dans r alors
                    merreur (b, double-définition)
                    E := E U {incrément (père (b))}
            fin
        sinon
            si a "masque" une déclaration b de l'identificateur dans une région
                    englobant r alors
                pour toute utilisation c de b faire
                  si région (c) est dans r alors E := E U {incrément (c)}
                fin
            fin
            si erreurl ≠ correct alors erreur := erreur-interne
                                   sinon erreur := correct
        fin
        merreur (a, erreur)
        mbase (a, base)
    fin
```

Remarque : pour alléger les algorithmes, nous n'avons pas montré le traitement effectué pour déterminer le type d'un identificateur (constante, type, variable, procédure, etc...).

CHAPITRE 5

CONCLUSION

5.1. EVALUATION DU PROTOTYPE.

Nous avons exposé dans les chapitres précédents les principes de réalisation de l'éditeur syntaxique et de la RI de l'atelier Adèle. Cette réalisation nous semble satisfaisante pour un prototype, mais un effort considérable reste à faire pour la rendre utilisable par d'autres personnes que celles ayant travaillé dans le projet. Nos critiques sont principalement basées sur l'aspect externe de la réalisation.

La conception de l'interface usager n'était pas terminée quand nous avons écrit l'éditeur. C'est pourquoi nous avons dû concevoir une interface très simple, permettant de valider les spécifications fonctionnelles de l'éditeur. De même, le décompilateur est peu satisfaisant dans l'optique d'une "bonne" version de l'atelier : l'écran est conceptuellement géré comme une suite infinie de lignes, puisque chaque commande provoque le réaffichage complet de la partie du programme concernée. L'utilisation de vitesses de transmission trop lentes (1200 bauds) provoque donc un délai gênant dans l'interaction.

La partie de programme décompilée est systématiquement l'unité englobant l'unité courante, permettant ainsi d'afficher l'environnement immédiat du centre d'intérêt de l'utilisateur à un instant donné. Mais cela nous semble inadéquat : ou bien le contexte ainsi affiché est insuffisant, comme dans le cas où l'unité englobante est une instruction simple, tenant sur une ligne ; ou bien il est en partie inutile, comme dans le cas où l'unité englobante est une instruction composée tenant sur plusieurs pages. Nous pensons qu'il est possible de tirer un meilleur profit de l'espace d'affichage disponible, par exemple en décompilant partiellement les unités soeurs de l'unité courante, ainsi que ses unités ancêtres : le contexte affiché doit être à la fois parental et latéral.

Les commandes de l'éditeur se révèlent dans l'ensemble satisfaisantes. La notion de zone de travail rejoint celle de marqueur utilisée dans Mentor, et permet des modifications puissantes ; de plus, elle apporte une vision homogène des unités prédéfinies et construites.

L'opérateur d'annulation nous semble une commande indispensable qui soulage l'utilisateur de la crainte d'effectuer par erreur des modifications irréversibles. L'utilisateur débutant est aussi encouragé à faire des essais de modification, et l'apprentissage de l'éditeur est ainsi facilité.

Quelques points sont susceptibles d'améliorations notables : les commandes de positionnement, la définition de l'opérateur de copie, les opérations de recherche, et la visualisation des erreurs.

Nous pensons que les commandes de déplacement d'un éditeur syntaxique doivent être beaucoup plus dirigées par la représentation textuelle du programme que par sa représentation arborescente. Par exemple, le déplacement de base pourrait être un parcours linéaire des "atomes" du programme, un atome étant le plus souvent une expression complète. L'utilisation répétée de cette commande, simplement en maintenant enfoncée une touche du clavier, peut aboutir à des déplacements extrêmement rapides dans le programme, à condition bien sûr que la machine réagisse suffisamment vite. Ces déplacements sont un peu similaires à ceux que l'on rencontre dans les éditeurs de textes "plein écran". Les "grands" déplacements dans le programme se font à l'aide de la commande de recherche des principales subdivisions d'un programme, comme par exemple une procédure ou une fonction. Enfin, la désignation d'une unité structurée englobant un atome s'effectue très simplement par remontée dans l'arbre : il est toujours plus facile de monter que de descendre.

L'opération de copie d'une unité nécessite la présence d'un emplacement vide. Ceci aboutit actuellement à la multiplication d'emplacements vides parasites, particulièrement dans les listes, que l'utilisateur doit détruire explicitement par une commande de nettoyage. Il manque donc une destruction implicite de ces emplacements, par exemple mise en oeuvre à l'appel de toute commande de déplacement. Mais peut-être vaut-il mieux revenir à des commandes plus classiques d'insertion avant ou après une unité, ce qui élimine le problème. Il serait possible de rendre implicite l'opérateur d'insertion après l'unité courante : la frappe du nom d'une unité serait interprétée soit comme une copie de l'unité après l'unité courante, soit comme le remplacement de l'unité courante si une seule unité est permise dans l'emplacement. De cette manière, seule l'insertion avant une unité serait explicite, et la frappe des opérateurs de recopie resterait globalement minime.

La commande de recherche d'une unité syntaxique nous semble suffisamment puissante. Mais il lui manque un mécanisme associé de réécriture pour exploiter pleinement le potentiel de l'édition syntaxique. Plus généralement, l'utilisateur ne dispose d'aucun moyen d'écrire de nouvelles commandes adaptées à ses besoins particuliers. Nous pensons qu'un bon éditeur syntaxique devrait posséder de telles facilités, et c'est pourquoi nous regrettons leur absence dans la plupart des éditeurs existants. A notre connaissance, seul Mentor dispose de cette faculté : le langage Mentol, malgré ses imperfections, est un remarquable outil de manipulation de programmes.

Enfin, les erreurs détectées par l'analyseur contextuel ne sont pas suffisamment prises en compte. Certes, elles sont mises en évidence par le décompilateur qui affiche en surbrillance les unités incorrectes ; mais l'utilisateur n'a aucun moyen de les désigner afin d'obtenir un diagnostic plus explicite et plus détaillé. Plus généralement, nous n'avons pas exploré tous les avantages que procure l'analyse incrémentale, qui devrait permettre la définition de commandes "contextuelles" : la visualisation du graphe d'appel des procédures et de la déclaration d'une variable en sont des exemples simples. Nous pensons que l'apport le plus significatif de l'édition syntaxique réside justement dans la possibilité de définir de telles commandes, qui facilitent réellement le processus d'écriture et de mise au point d'un programme, et justifient amplement l'effort de développement d'un analyseur incrémental.

Examinons maintenant la réalisation effectuée d'un point de vue interne, et notamment le taux d'expansion de la RI et l'implémentation des procédures de déroutement.

Avec les annotations actuellement définies, l'arbre d'un programme Pascal prend entre quatre et cinq fois plus de place en mémoire centrale que le texte source équivalent. Ce rapport tombe entre un et deux en mémoire secondaire, car les "pointeurs" ne sont pas conservés du fait de l'archivage de l'arbre sous forme préfixée. Ces rapports tendent à diminuer lorsque la taille du programme augmente, principalement à cause des identificateurs et des commentaires. Et nous pensons qu'il est facile de trouver une implémentation moins gourmande en mémoire.

Plus significatif est le fait que la taille de la RI dépend pour une bonne part des annotations qui sont définies. Puisque chaque outil est susceptible d'apporter son propre jeu d'annotations, il est particulièrement important d'avoir un mécanisme de définition dynamique des annotations : une définition statique impliquerait une réservation systématique de mémoire à la création d'un noeud, et le taux d'expansion deviendrait à terme démesuré.

Enfin, les procédures de déroutement n'ont pas pu être implantées dans la couche basse de gestion de la RI, à cause de la déclaration statique des procédures en Pascal. C'est pourquoi la couche haute gère explicitement l'ordre d'appel à ces procédures. Il en résulte que cette couche doit être reconsidérée lors de l'ajout ou de la modification d'un outil dans l'atelier. En particulier, il est important d'examiner soigneusement l'ordre des appels qui est fonction de la dépendance entre outils. Par exemple, l'appel au décompilateur, qui utilise l'annotation erreur, doit s'effectuer après l'appel à l'analyseur, qui met à jour cette annotation.

Deux améliorations seraient possibles : d'une part étudier comment spécifier les actions d'un outil sur la RI, afin d'en déduire des dépendances et un ordre d'appel des procédures de déroutement ; et d'autre part étendre cette notion de déroutement à la modification des annotations, ce qui résoudrait trivialement la dépendance analyseur-décompilateur. Ceci permettrait alors de définir un atelier configurable dynamiquement, où chaque outil se "présenterait" à la couche de gestion de la RI au moment de son activation, et apporterait son jeu de procédures de déroutement. De cette manière, les outils coopèreraient sans se connaître mutuellement.

5.2. PERSPECTIVES.

Nous avons essayé dans cette thèse de dégager comment la notion de programme pouvait être perçue dans un atelier de développement de logiciel. Cette étude a permis de montrer l'importance du choix d'une représentation des programmes : les différents outils ont ainsi une vision homogène d'un programme et participent plus harmonieusement au processus d'écriture d'un logiciel.

La notion de syntaxe abstraite est certainement la plus appropriée pour décrire et représenter un programme, et joue bien le rôle fédérateur exigé d'une RI. Les annotations permettent d'adapter la représentation commune aux besoins de chaque outil.

Nous avons développé la notion de rameau, qui aboutit à une représentation plus adaptée que la simple syntaxe abstraite utilisée dans les systèmes existants : elle permet une représentation plus synthétique de la notion de liste d'unités syntaxiques. Mais cette notion n'est pas encore totalement exploitable, car elle complique l'expression de certains traitements sur l'arbre (par exemple la description d'un décompilateur ou d'une grammaire abstraite attribuée).

C'est cependant sur la notion elle-même de syntaxe abstraite que cette partie de notre étude nous laisse un sentiment d'inachevé. Nous avons signalé que différents concepteurs définiraient sans doute différentes grammaires abstraites pour le même langage de programmation. Ces différences proviennent d'une part de l'idée que se fait le concepteur de l'utilisation de l'arbre dans un atelier, et d'autre part de la difficulté de bien discerner les informations profondes d'un programme (par opposition aux informations de surface, de représentation concrète). Si les langages étaient conçus dans l'ordre syntaxe abstraite et sémantique, puis seulement syntaxe concrète, alors le problème du choix de la syntaxe abstraite disparaîtrait. Ceci semble la tendance actuelle des concepteurs de langages, mais l'accord sera toujours difficile entre le concepteur de syntaxe abstraite et le réalisateur d'un outil.

Un autre aspect de notre travail a été l'étude des concepts permettant une analyse contextuelle incrémentale d'un programme. Nous avons montré qu'un formalisme tel qu'une grammaire attribuée est la base de tels mécanismes. D'autres formalismes tels que les grammaires affixes, les grammaires attribuées étendues et les grammaires de conjugaison, permettent également d'exprimer la syntaxe contextuelle d'un langage, et il serait intéressant d'étudier s'ils sont plus

adaptés au problème de la génération automatique d'analyseurs incrémentaux.

Quoi qu'il en soit, notre étude a également montré qu'il reste beaucoup de travail avant de pouvoir produire automatiquement des analyseurs incrémentaux réellement satisfaisants. La méthode de Johnson et Fischer paraît très prometteuse, mais ne débouchera que difficilement sur un générateur d'analyseurs incrémentaux, car elle comprend des traitements ad hoc difficilement généralisables. Mais, en attendant mieux, elle fournit une démarche très profitable pour l'écriture manuelle d'un analyseur pour un langage donné. L'exemple d'Adèle montre que la complexité des algorithmes reste sensiblement la même qu'un analyseur contextuel non incrémental.

Les techniques d'analyse incrémentale pourraient servir à résoudre plus de problèmes que la seule analyse contextuelle d'un programme. Deux applications en seraient déjà possibles dans l'atelier Adèle : la base de programmes et le compositeur. En effet, dans ces deux cas il existe le besoin d'effectuer un traitement lors de toute modification des données manipulées. Pour le service d'archivage, il est nécessaire de connaître la liste de dépendances entre modules, afin de calculer les conséquences de toute modification d'un composant. Pour le compositeur, l'interprétation d'une modification de l'arbre de boîtes doit être incrémentale, afin de ne réafficher que le strict minimum (signalons à ce sujet que Reps illustre justement son approche par l'affichage d'un texte structuré, ce qui montre bien la similitude des problèmes).

Une étude très profitable serait donc l'approfondissement des mécanismes incrémentaux, afin de permettre leur utilisation dans tous les domaines où l'interprétation interactive d'une modification est nécessaire.

Finalement, la partie la plus discutable de notre travail est l'étude des spécifications externes d'un éditeur syntaxique. Il est difficile de dire si telle fonction est préférable à telle autre, indépendamment de la manière dont l'utilisateur peut l'appeler. En particulier, nous ne savons pas s'il faut des commandes purement syntaxiques, ou s'il faut autoriser des commandes plus "textuelles". Peut-être l'idéal est un éditeur mixte, syntaxique et textuel, permettant à l'utilisateur de choisir à tout instant la vision adaptée à un problème particulier.

Il nous semble difficile de prédire à quoi ressembleront les futurs éditeurs de programmes : après tout, l'édition syntaxique en est encore à sa première génération, et ses concepts ne sont pas aussi mûrs que ceux de

l'édition textuelle. Aussi, cette dernière est et restera encore longtemps plus attrayante. Mais malgré son "péché de jeunesse", qui fausse la concurrence entre les deux types d'éditeurs, nous pensons que l'édition syntaxique est promise à un brillant avenir.

Un atelier de génie logiciel ne se limite pas à l'écriture d'un programme, et c'est pourquoi notre apport reste limité. Mais nous pensons que notre travail s'intègre bien au vaste mouvement qui pousse à concevoir des ateliers. Edition syntaxique incrémentale figurera sans doute parmi les maîtres mots des futurs ateliers, à côté de modularité, interprétation et interface usager.

ANNEXE 1 : COMMANDES DE L'EDITEUR.

v : Visualisation de(s) commande(s) disponible(s).
Sans arguments, cette commande permet de lister l'ensemble des commandes, ainsi qu'un résumé de leurs fonctions.
v <com>, suivi d'un nom de commande, permet d'obtenir des explications plus détaillées sur l'utilisation et les fonctions de la commande concernée.
v <nomzt>, suivi d'un nom de zone de travail, permet de visualiser le contenu de cette zone, en la décompilant, sans avoir à s'y positionner.
v <predef>, suivi d'un nom d'unité syntaxique prédéfinie, permet de visualiser la structure de cette unité.

e : Ecriture du programme dans un fichier.
Sans arguments, cette commande permet l'écriture dans le dernier fichier lu.
e <nomfich>, suivi d'un nom de fichier : "nomfich" ou "nomfich.ri", l'écriture se fait dans le fichier "nomfich.ri".

l : Lecture d'un programme dans un fichier.
l <nomfich> (valable seulement si la zone de travail courante est la zone principale), permet de lire le fichier "nomfich.ri".

pre : Enumération des noms d'unités syntaxiques prédéfinies.
Ces unités syntaxiques peuvent être directement introduites à l'endroit courant du programme, en tapant leurs noms.

def : Enumération des noms de zone de travail actuellement définies.
En plus de la zone de travail principale, l'utilisateur peut créer d'autres zones de travail (commande : "cr"), qui pourront être référencées par leur nom, indiqué à la création.

cl : Enumération des noms de classes d'unités syntaxiques.
Cette commande permet de lister les noms de classes d'unités syntaxiques utilisables pour la recherche d'un modèle (commande "t").

pos : Positionnement direct dans une zone de travail.
pos <n>, suivi d'un entier positif "n", permet de se positionner sur la ligne de numéro "n" de la zone de travail courante (dans ce cas, l'opérateur "pos" est facultatif, et, s'il n'est pas utilisé, "n" est

obligatoire) ; par défaut, n vaut 1 quand "pos" est utilisé.

pos <nomzt>, suivi d'un nom de zone de travail non prédéfinie, permet de se positionner au début de cette zone de travail.

sup : Passage à un niveau syntaxique supérieur (englobant).

sup <n>, suivi d'un entier positif "n", permet de "remonter" de "n" niveaux syntaxiques par rapport à la position courante (par défaut, n vaut 1).

inf : Passage à un niveau syntaxique inférieur.

inf <n>, suivi d'un entier positif "n", permet de "descendre" sur la n-ième unité par rapport à la position courante (par défaut, n vaut 1).

suiv : Passage à une unité syntaxique "suivante", de même niveau.

suiv <n>, suivi d'un entier "n", permet un déplacement dans le niveau syntaxique courant (par défaut, n vaut 1).

Si n=0, on reste sur la position courante ; si n>0, on passe à la n-ième unité suivant la position courante ; si n<0, on passe à la n-ième unité à partir du début du niveau courant.

prec : Passage à une unité syntaxique "précédente", de même niveau.

prec <n>, suivi d'un entier "n", permet un déplacement dans le niveau syntaxique courant (par défaut, n vaut 1).

Si n=0, on reste sur la position courante ; si n>0, on passe à la n-ième unité précédant la position courante ; si n<0, on passe à la n-ième unité syntaxique à partir de la fin du niveau courant.

t : Recherche d'un modèle.

t <nomzt>, suivi d'un nom de zone de travail définie ou prédéfinie, permet d'en trouver une occurrence dans la zone de travail courante.

t <classe>, suivi d'un nom de classe, permet de rechercher la première occurrence d'un unité syntaxique de cette classe.

mq : Marquage (repérage) de l'unité syntaxique courante.

Cette commande permet de repérer une unité syntaxique afin de pouvoir la désigner ultérieurement sans avoir à s'y positionner ; sa désignation pourra être faite en indiquant le nom "marq" comme s'il s'agissait d'un nom de zone de travail dans laquelle on a copié une unité syntaxique.

rmq : Désignation (repérage) d'un ensemble d'unités syntaxiques.

Ayant précédemment "marqué" une unité syntaxique (commande "mq"), cette commande sert à délimiter un ensemble d'unités de même niveau syntaxique ; l'ensemble ainsi repéré par le nom "marq" sera alors constitué des unités syntaxiques comprises entre la première marque et l'unité courante.

dmq : Démarquage d'une ou d'un ensemble d'unités syntaxiques.
Cette commande annule l'effet de repérage des commandes "mq" et "rmq" précédemment exécutées.

cr : Création d'une zone de travail.
cr <chaîne>, suivi d'une chaîne de caractères, permet de créer une zone de travail qui pourra être référencée par le nom "chaîne".

d : Destruction d'une unité syntaxique.
Sans indication supplémentaire, cette commande détruit l'unité syntaxique courante, celle-ci étant remplacée par l'unité "vide".
d <nomzt>, suivi d'un nom de zone de travail, permet de détruire le contenu de la zone de travail désignée.

i : Insertion avant l'unité syntaxique courante.
Cette commande est utilisable pour "étendre" un ensemble d'unités syntaxiques de même niveau ; son effet est d'insérer l'unité "vide" avant l'unité courante. Ainsi, pour rajouter une instruction "if", dans une liste d'énoncés, avant l'énoncé courant, on peut taper : "i ; if".

a : Ajout après l'unité syntaxique courante.
Cette commande est utilisable pour "étendre" un ensemble d'unités syntaxiques de même niveau ; son effet est d'ajouter l'unité "vide" après l'unité courante. Ainsi, pour rajouter une instruction "if", dans une liste d'énoncés, après l'énoncé courant, on peut taper : "a ; if".

dep : Déplacement (transfert) d'une unité syntaxique à l'endroit courant.
dep <nomzt>, suivi d'un nom de zone de travail, permet de transférer le contenu de cette zone à l'endroit courant (par défaut, le nom de zone de travail est "marq").

cp : Copie (duplication) d'une unité syntaxique à l'endroit courant.
cp <nomzt> ou cp <predef>, suivi d'un nom de zone de travail (ou d'un nom d'unité syntaxique prédéfinie), permet de copier le contenu de la zone

de travail (ou l'unité syntaxique prédéfinie) à l'endroit courant. (l'opérateur "cp" est facultatif).

$: Passage en mode introduction syntaxique.
Cette commande permet d'introduire une (ou plusieurs) unité(s) syntaxique(s) à l'endroit courant, sous forme de texte source Pascal ; terminer par "çf".

mx : Passage au niveau de commande Multics.
Cette commande permet de revenir au niveau de commande Multics sans sortir de l'éditeur ; taper "q" pour revenir à l'éditeur.

f : Fin de l'édition.
Actuellement, la sortie de l'éditeur se fait sans vérification d'écriture.

net : Nettoyage de la zone de travail courante, à partir de l'endroit courant. Cette commande permet de supprimer tous les emplacements "vide" inutiles situés dans les niveaux inférieurs au niveau courant, dans la zone de travail courante.

num : Renumérotage des lignes.
Cette commande permet de réorganiser le numérotage des lignes de toutes les zones de travail (zone de travail principale comprise).

ren : Renommer l'identificateur courant.
ren <chaîne>, permet de changer le nom de l'identificateur repéré par le sommet courant, en lui donnant le nouveau nom "chaîne".

dec : Décompilation.
Cette commande permet l'affichage, sous forme de texte source Pascal, de l'unité syntaxique courante.

anl : Annulation.
Cette commande permet d'annuler la dernière commande "annulable" exécutée, sauf si cette commande a été exécutée depuis.
- commandes annulables : pos, sup, inf, suiv, prec, t, mq, rmq, dmq, d, i, a, dep, cp, l, $;
- commandes non annulables : net, num, ren, anl, I ;
- commandes sans effet sur la sauvegarde : les autres.

I : Commande réservée.

I <com>, suivi d'un nom de commande parmi les suivantes, permet de manipuler des fonctions de test qui sont appelées à disparaître :

- s : passage en mode "sans analyse" ;
- a : passage en mode "analyse contextuelle" ;
- v : valeur des annotations du sommet courant ;
- i : imprimer le sous-arbre courant ;
- j <n> : jump à l'adresse "n" ;
- m <a> <v> : affecter la valeur v à l'annotation a du sommet courant.

Si vous désirez des détails sur une commande "com", tapez : "v com" ;
Il est possible d'enchaîner des commandes sur une même ligne, en les séparant par des point-virgules.

ANNEXE 2 : GRAMMAIRE ABSTRAITE DU LANGAGE PASCAL.

A2.1) Notations.

Les phylums sont décrits sous la forme X ::= X1, X2, ... Xn. Les opérateurs sont sous la forme t ==> U1, U2, ..., Un (n>0), où les Ui peuvent être :

- X (rameau simple),
- <X> (rameau optionnel),
- X* (rameau composé d'un nombre quelconque d'éléments),
- X+ (rameau composé d'au moins un élément).

Après chaque opérateur se trouve une description informelle de sa représentation concrète. Les nombres utilisés correspondent aux rameaux de même rang. Les séparateurs d'éléments d'un rameau de type liste ne sont pas représentés. Les rameaux optionnels sont représentés comme s'ils étaient simples.

A2.2) Phylums.

TOUS ::= DEBUT, CONSTANTE, DESCRIPTEUR, PROCFONCT, PARAMETRE, PAREFFECTIF, ENONCE, REFERENCE, EXP, PREDEF, DIVERS

DEBUT ::= prog

CONSTANTE ::= plusconst, moinsconst, constchaîne, constnil, CONSTSIMPLE

CONSTSIMPLE ::= constentier, constreel, ident

DESCRIPTEUR ::= ident, pointeur, TYPORDINAL, TYPESTRUCTURE

TYPORDINAL ::= ident, enum, intervalle

TYPESTRUCTURE ::= tableau, ensemble, fichier, article

PROCFONCT ::= declproc, declfonct

PARAMETRE ::= parvaleur, parvariable, parproc, parfonct

PAREFFECTIF ::= ident, writepar, REFERENCE, EXP

ENONCE ::= appelproc, affect, choix, allera, sialors, répéter, tantque, avec, pourt, pourd, début

REFERENCE ::= ident, accestab, acceschamp, indirect

EXP ::= REFERENCE, CONSTANTE, OPBINAIRE, OPUNAIRE, appelfonct

OPBINAIRE ::= plus, moins, mult, divreel, divent, modulo, et, ou, égal, dif, inf, infeg, sup, supeg, dans

OPUNAIRE ::= plusun, moinsun, non

```
/* opérateurs prédéfinis */
PREDEF        ::= PRECONST, PRETYPE, PREPROC, PREFONC
PRECONST      ::= premaxint, pretrue, prefalse
PRETYPE       ::= preinteger, prereal, preboolean, prechar, pretext
PREPROC       ::= predispose, preget, prenew, prepack, prepage, preput,
                  preread, prereadln, prereset, prerewrite, preunpack,
                  prewrite, prewriteln
PREFONCT      ::= preabs, prearctan, prechr, precos, preeof, preeoln, preexp,
                  preln, preodd, preord, prepred, preround, presin, presqr,
                  presqrt, presucc, pretrunc
DIVERS        ::= declconst, decltype, declvar, packe, champ, partievariante,
                  variante, elemchoix.
```

A2.3) OPERATEURS.

```
racine        ==> TOUS *, DEBUT*
                  /* non décompilé */

prog          ==> ident, ident*, constent*, declconst*, decltype*, declvar*,
                  PROCFONCT*, ENONCE*
                 /* program 1 (2) ;
                      label 3 ;
                      const 4 ;
                      type  5 ;
                      var   6 ;
                       7 ;
                      begin  8  end.   */

ident         ==>
                 /* décompilation de l'annotation string */

constent      ==>
                 /* décompilation de l'annotation string */

constreel     ==>
                 /* décompilation de l'annotation string */

constchaîne   ==>
                 /* décompilation de l'annotation string */
```

```
constnil        ==>
                   /* nil */

plusconst       ==> CONSTSIMPLE
                   /*  + 1  */

moinsconst      ==> CONSTSIMPLE
                   /*  - 1  */

declconst       ==> ident, CONSTANTE
                   /*  1 = 2  */

decltype        ==> ident, DESCRIPTEUR
                   /*  1 = 2  */

declvar         ==> ident+, DESCRIPTEUR
                   /*  1 : 2  */

declproc        ==> ident, PARAMETRE*, constent*, declconst*, decltype*,
                    declvar*, PROCFONCT*, ENONCE*
                   /* procédure  1(2) ; label 3 ; const 4 ; type 5 ; var 6 ;
                                 7 ; begin 8 end ;  */

declfonct       ==> ident, PARAMETRE*, ident, constent*, declconst*, decltype*,
                    declvar*, PROCFONCT*, ENONCE*
                   /* function  1(2) : 3 ; label 4 ; const 5 ; type 6 ; var 7 ;
                                 8 ; begin 9 end ;  */

pointeur        ==> TYPORDINAL
                   /* ↑ 1  */

enum            ==> ident+
                   /*  (1)  */

intervalle      ==> CONSTANTE, CONSTANTE
                   /*  1 .. 2  */

tableau         ==> <packe>, TYPORDINAL+, DESCRIPTEUR
                   /*  1 array [2] of 3  */
```

```
ensemble        ==> <packe>, TYPORDINAL
                    /*  1 set of 2  */

fichier         ==> <packe>, DESCRIPTEUR
                    /*  1 file of 2  */

article         ==> <packe>, champ*, <partievariante>
                    /*  1 record 1 ; 2 end  */

champ           ==> ident+, DESCRIPTEUR
                    /*  1 : 2  */

partievariante==> <ident>, ident, variante+
                    /*  case 1 : 2 of 3  */

variante        ==> CONSTANTE+, champ*, partievariante
                    /*  1 : (2 ; 3)  */

packe           ==>
                    /*  packed  */

parvaleur       ==> ident+, ident
                    /*  1 : 2  */

parvariable     ==> ident+, ident
                    /*  var 1 : 2  */

parproc         ==> ident, PARAMETRE*
                    /*  procédure 1 (2)  */

parfonct        ==> ident, PARAMETRE*, ident
                    /*  function  1 (2) : 3  */

appelproc       ==> ident, PAREFFECTIF*
                    /*  1 (2)  */

writepar        ==> EXP, <EXP>, <EXP>
                    /*  1 : 2 : 3  */
```

```
appelfonct    ==> ident, EXP*
                  /*  1 (2)  */

affect        ==> <constent>, REFERENCE, EXP
                  /*  1 : 2 := 3  */

choix         ==> <constent>, EXP, elemchoix+
                  /* 1 : case 2 of 3 end  */

elemchoix     ==> CONSTANTE+, ENONCE
                  /*  1 : 2  */

allera        ==> <constent>, constent
                  /*  1 : goto 2  */

sialors       ==> <constent>, EXP, ENONCE, ENONCE
                  /*  1 : if 2 then 3 else 4  */

repeter       ==> <constent>, ENONCE*, EXP
                  /*  1 : repeat 2 until 3  */

tantque       ==> <constent>, EXP, ENONCE
                  /*  1 : while 2 do 3  */

avec          ==> <constent>, REFERENCE+, ENONCE
                  /*  1 : with 2 do 3  */

pourt         ==> <constent>, ident, EXP, EXP, ENONCE
                  /*  1 : for 2 := 3 to 4 do 5  */

pourd         ==> <constent>, ident, EXP, EXP, ENONCE
                  /*  1 : for 2 := 3 downto 4 do 5  */

début         ==> <constent>, ENONCE*
                  /*  1 : begin 2 end  */

accestab      ==> REFERENCE, EXP+
                  /*  1 [2]  */
```

```
acceschamp    ==> REFERENCE, ident
                  /*  1 . 2  */

indirect      ==> REFERENCE
                  /*   1↑ */

plus          ==> EXP, EXP
                  /*  1 + 2  */

moins         ==> EXP, EXP
                  /*  1 - 2  */

mult          ==> EXP, EXP
                  /*  1 * 2  */

divreel       ==> EXP, EXP
                  /*  1 / 2  */

divent        ==> EXP, EXP
                  /*  1 div 2  */

modulo        ==> EXP, EXP
                  /*  1 mod 2  */

et            ==> EXP, EXP
                  /*  1 and 2  */

ou            ==> EXP, EXP
                  /*  1 or 2  */

égal          ==> EXP, EXP
                  /*  1 = 2  */

diff          ==> EXP, EXP
                  /*  1 <> 2  */

inf           ==> EXP, EXP
                  /*  1 < 2  */
```

```
infeg       ==> EXP, EXP
                /*  1 <= 2  */

sup         ==> EXP, EXP
                /*  1 > 2  */

supeg       ==> EXP, EXP
                /*  1 >= 2  */

dans        ==> EXP, EXP
                /*  1 in 2  */

plusun      ==> EXP
                /*  + 1  */

moinsun     ==> EXP
                /*  - 1  */

non         ==> EXP
                /*  not 1  */
```

BIBLIOGRAPHIE

<Ada 80a>
"Reference Manual for the Ada programming language"
Proposed standard document, CII Honeywell Bull, Juillet 1980.

<Ada 80b>
"Formal definition of the Ada programming language"
Preliminary version for public review, INRIA, Novembre 1980.

<André 82>
André E.
"Présentation du projet pilote CONCERTO"
Journées Bigre 82, Grenoble, Janvier 1982, pp. 231-242.

<Atkinson 78>
Atkinson L., Mac Gregor L.
"CONA - A Conversational Algol System"
Software Practice and Experience, Vol. 8, 1978, pp. 699-708.

<Atkinson 81>
Atkinson L., North D.
"COPAS - A Conversational Pascal System"
Software Practice and Experience, Vol. 11, 1981, pp. 819-829.

<Berlioux 72>
Berlioux P.
"Etude d'automates et de grammaires de ramifications"
Thèse 3ème Cycle, Grenoble, Juillet 1972.

<Berthaud 73>
Berthaud M., Griffiths M.
"Incremental compilation and conversational interpretation"
Annual Review of Automatic Programming, Vol. 7(2), 1973, pp. 95-114.

<Briat 81> (Adèle)
Briat J., Cheval J.L., Krakowiak S., Laforgue P., Mossière J., Raymond J., Rouzaud Y.
"ADELE : un atelier de développement de génie logiciel"
Congrès AFCET, Gif-sur-Yvette, Novembre 1981, pp. 181-199.

<Bouchenez 80>
Bouchenez J.L., Loyer M., Maurice P., Prusker F., Sogno J.C., Vercoustre A.M.
"Le système LEGOS : environnement de programmation sur Mitra 125"
Journées Bigre 80, Rennes, Décembre 1980, pp. 19-22.

<Brosgol 80>
Brosgol B., Newcomer J., Lamb D., Levine D., Densen M., Wulf W.
"TCOL-Ada : Revised report on an intermediate representation for the preliminary Ada language"
TR CS-80-105, Carnegie-Mellon University, Février 1980.

<Bruandet 76>
Bruandet M.F.
"Optimisation d'un compilateur incrémental et conversationnel de PL/1"
Thèse 3ème cycle, Grenoble, Juillet 1976.

<Conchon 83>
Conchon A., Lang B.
"Spécification d'un noyau de manipulation d'arbres pour un environnement de programmation"
Journées Bigre 83, Cap d'Agde, Octobre 1983, pp. 617-628.

<Coutaz-Raymond 83> (Adèle)
Coutaz-Raymond J., Herrmann M.
"Adèle et le médiateur-compositeur, ou comment rendre une application interactive indépendante de l'interface usager".
Journées Bigre 83, Cap d'Agde, Octobre 1983, pp. 1-17.
Deuxième Colloque de Génie Logiciel (AFCET), Nice, Juin 1984 (à paraître).

<Cheval 82> (Adèle)
Cheval J.L., Estublier J., Ghoul S., Krakowiak S.
"Modularité et composition des programmes dans l'atelier de logiciel Adèle"
Premier Colloque de Génie Logiciel (AFCET), Paris, Juin 1982, pp. 183-197.

<Cheval 83> (Adèle)
Cheval J.L., Estublier J., Ghoul S.
"Un système automatique de gestion de versions : la base de programmes Adèle"
Journées Bigre 83, Cap d'Agde, octobre 1983, pp. 481-494.

<Cheval 84> (Adèle)
Cheval J.L.
"Interface d'utilisation d'une base de programmes"
Deuxième Colloque de Génie Logiciel (AFCET), Nice, Juin 1984 (à paraître)

<De Remer 80>
De Remer F., Jullig R.
"Tree-affix dendrogrammars for languages and compilers"
Lecture Notes in Computer Science n° 94, Springer-Verlag, 1980, pp. 300-319.

<DOD 80>
"Stoneman : requirements for Ada Programming Support Environments"
US Department of Defense, Buxton J. editor, Février 1980.

<Donzeau-Gouge 75>
Donzeau-Gouge V., Huet G., Kahn G., Lang B., Lévy J.J.
"A structure-oriented program editor : a first step towards computer-assisted programming"
Rapport IRIA 114, Avril 1975.

<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 80a>
Donzeau-Gouge V., Huet G., Kahn G., Lang B.
"Programming environments based on structured editors : the MENTOR experience"
Rapport INRIA 26, Juillet 1980.

<Donzeau-Gouge 80b>
Donzeau-Gouge V., Kahn G., Lang B.
"On the formal definition of Ada"
Lecture Notes in Computer Science n° 94, Springer-Verlag, 1980, pp. 475-489.

<Donzeau-Gouge 83>
Donzeau-Gouge V., Kahn G., Lang B., Mélèse B.
"Outline of a tool for document manipulation"
IFIP Congress, Paris, Septembre 1983.

<Estublier 83> (Adèle)
Estublier J., Krakowiak S., Mossière J., Rouzaud Y.
"Design principles of the Adèle programming environment"
International Computing Symposium (ACM), Nuremberg, Mars 1983.

<Estublier 84a> (Adèle)
Estublier J., Ghoul S., Krakowiak S.
"Preliminary experience with a configuration control system for modular programs"
ACM Sigplan/Sigsoft Symposium on Practical Software Development Environments, Pittsburgh, Avril 1984 (à paraître).

<Estublier 84b> (Adèle)
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 (à paraître).

<Gandalf 82>
Habermann N., Ellison R., Medina-Mora R., Feiler P., Notkin D., Kaiser G., Garlan D., Popovitch S.
"The second Compendium of Gandalf Documentation"
TR CS-82, Carnegie-Mellon University, Mai 1982.

<Gasich 81>
Gasich D.B.
"Qualité des programmes, productivité de la programmation et génie logiciel"
Congrès AFCET, Gif-sur-Yvette, Novembre 1981, pp. 3-24.

<Ghoul 83> (Adèle)
Ghoul S.
"Base de données et gestion de configurations dans un atelier de génie logiciel"
Thèse Docteur-Ingénieur, Grenoble, Décembre 1983.

<Goos 81>
Goos G., Wulf W.M.
"Diana referenoe manual"
TR CS-81-01, Carnegie-Mellon University, Mars 1981.

<Guerrier 80>
Guerrier J.
"Utilisation du système Mentor comme outil de transport"
Journées Bigre 80, Rennes, Décembre 1980, pp. 35-40.

<Habermann 79>
Habermann N.
"The Gandalf research project"
CS research review, Carnegie-Mellon University, 1979.

<Hansen 71>
Hansen W.
"Creation of hierarchic text with a computer display"
PhD Thesis, Stanford University, Juin 1971.

<Hausen 81>
Hausen M.L., Mullerburg M.
"Conspectus of software engineering environments"
International Computing Symposium (ACM), San Diego, Mars 1981.

<Herrmann 84> (Adèle)
Herrmann M.
"Interface usager-application dans un atelier de génie logiciel"
Thèse 3ème Cycle, Grenoble, 1984 (à paraître).

<Hulot 83>
Hulot J.M.
"Ceyx, a multiformalism environment"
Rapport INRIA 210, Mai 1983.

<Johnson 82>
Johnson G., Fischer C.
"Non-syntactic attribute flow in language based editors"
9th ACM Symposium on Principles of Programming Languages, Janvier 1982, pp. 185-195.

<Knuth 68>
Knuth D.
"Semantics of context-free languages"
Math. Systems Theory, Vol. 2(2), Juin 1968, pp. 127-147.
Math. Systems Theory, Vol. 5(1), Mars 1971, pp. 95-96.

<Krakowiak 82>
Krakowiak S.
"Systèmes intégrés de production de logiciel : concepts et réalisations"
Technique et Science Informatique, Vol. 1(3), 1982, pp. 187-200.

<Krakowiak 83> (Adèle)
Krakowiak S., Mossière J.
"Atelier de développement de logiciel pour la gestion de programmes modulaires et la génération de code multicibles"
Rapport final, Convention ADI 80-225, Avril 1983.

<Lenne 84> (Adèle)
Lenne C.
"Interprétation et mise au point de programmes dans un atelier de génie logiciel"
Thèse 3ème Cycle, Grenoble, 1984 (à paraître).

<Lorho 74>
Lorho B.
"De la définition à la traduction des langages de programmation : méthode des attributs sémantiques"
Thèse d'état, Toulouse, 1974.

<Lucrèce 82>
Lucrèce L., Maurice P., Vercoustre A.M.
"Minerve : manuel de référence"
Rapport INRIA 7, Septembre 1982.

<Marchand 74>
Marchand P.
"Etude et classification des bigrammaires. Application à l'étude des systèmes transformationnels"
Thèse de Spécialité, Nancy, Novembre 1974.

<Massié 80>
Massié H.
"Etude et réalisation d'un éditeur de programmes pour le langage extensible Let"
Thèse de Spécialité, Toulouse, Février 1980.

<Medina-Mora 82>
Medina-Mora R.
"Syntax-directed editing : towards integrated programming environments"
PhD. Thesis Carnegie-Mellon University, Mars 1982.

<Mélèse 82>
Mélèse B.
"Métal : un langage de spécifications pour le système Mentor"
Technique et Science Informatiques, Vol. 1(4), 1982, pp. 275-286.

<Mossière 82> (Adèle)
Mossière J., Raymond J., Rouzaud Y.
"Représentation interne et manipulation de programmes dans l'atelier de logiciel Adèle"
Premier Colloque de Génie Logiciel (AFCET), Paris, Juin 1982, pp. 137-150.

<Nestor 81>
Nestor J., Wulf W.M., Lamb D.
"IDL : Interface Description Language"
TR CS-81-139, Carnegie-Mellon University, AoÛt 1981.

<Oppen 80>
Oppen D.
"Prettyprinting"
ACM Transactions on Programming Languages and Systems, Vol. 18(4), Avril 1983.

<Pair 68>
Pair C., Quéré A.
"Définition et étude des bilangages réguliers"
Information and Control, Vol. 13, 1968, pp. 565-593.

<Peccoud 68>
Peccoud M., Griffiths M., Peltier M.
"Incremental interactive compilation"
IFIP Congress, Edimburgh, 1968, pp. 384-387.

<Reps 81>
Reps T.
"The Synthesizer Editor Generator : reference manual"
Cornell University, Septembre 1981.

<Reps 82>
Reps T.
"Optimal-time incremental semantic analysis for syntax-directed editors"
9th ACM Symposium on Principles of Programming Languages and Systems, Janvier 1982, pp. 169-176.

<Reps 83>
"Incremental context-dependant analysis for language-based editors"
ACM Transactions on Programming Languages and Systems, Vol. 5(3), Juillet 1983, pp. 449-477.
Corrigenda : ACM TOPLAS, Vol. 5(4), Octobre 1983, p. 680.

<Sandewall 78>
Sandewall E.
"Programming in an interactive environment : the Lisp experience"
ACM Computing Surveys, Vol. 10(1), Mars 1978, pp. 35-71.

<Santana 83a> (Adèle)
Santana M., Hochain J.C.
"Gemme : un générateur de code multi-machines"
Journées Bigre 83, Cap d'Agde, Octobre 1983, pp. 470-480.

<Santana 83b> (Adèle)
Santana M.
"Un système de production automatique de générateurs de code"
Thèse 3ème Cycle, Grenoble, Décembre 1983.

<Shapiro 81>
Shapiro E., Collins G., Johnson L., Ruttenberg J.
"Pases : a programming environment for Pascal"
ACM Sigplan Notices, Vol. 16(8), Août 1981, pp. 50-56.

<Simonet 81>
Simonet M.
"W-grammaires et logique du premier ordre pour la définition et l'implantation des langages"
Thèse d'état, Grenoble, Juillet 1981.

<Teitelbaum 80a>
Teitelbaum T., Reps T.
"The Cornell Program Synthesizer : a syntax-directed programming environment"
Cornell University, Mai 1980.

<Teitelbaum 80b>
Teitelbaum T., Reps T., Horwitz S.
"The why and wherefore of the Cornell Program Synthesizer"
ACM Symposium on Text Manipulation, Portland, Juin 1980, pp. 8-16.

<Teitelbaum 81>
Teitelbaum T., Reps T.
"The Cornell Program Synthesizer"
Communications ACM, Vol. 24(10), Octobre 1981.

<Vercoustre 83>
Vercoustre A.M.
"Minerve : un méta-éditeur syntaxique"
Rapport INRIA 229, Juillet 1983.

<Waters 82>
Waters R.
"Program editors should not abandon text-oriented commands"
ACM Sigplan Notices, Vol. 17(7), Juillet 1982, pp. 39-46.
Notkin D., Habermann N., Ellison R., Kaiser G., Garlan D.
"Letter to the editor"
ACM Sigplan Notices, Vol. 18(4), Avril 1983.

ERRATA

P. 99 : dernière phrase du premier paragraphe 4.3.2 : "Dans ce cas, les attributs hérités reçoivent une valeur arbitraire", ajouter la phrase: "(ainsi que les attributs synthétisés qui en dépendent)".

P. 100 : lignes 3 à 5 : "affecter aux attributs ... devenus incohérents", remplacer "A2" par "A3" et "A3" par "A2".

P. 100 : dans le paragraphe : "L'initialisation de l'algorithme ...", supprimer le mot "hérités".

P. 101 : lignes 8 à 11 : "affecter aux attributs ... racine de A3", remplacer "A2" par "A3" et "A3" par "A2".

P. 102 : lignes 16 à 18 : "affecter aux attributs ... racine de A3", remplacer "A2" par "A3" et "A3" par "A2".

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de Messieurs

. S. KRAKOWIAK, Professeur

. E. ANDRE, Ingénieur

°

Monsieur ROUZAUD Yann

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 24 mai 1984

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,