HAL open science

Production de logiciels pour l'enseignement : une expérience de prototypage d'un système construit sur un environnement Prolog

Alain Lucci

▶ To cite this version:

Alain Lucci. Production de logiciels pour l'enseignement : une expérience de prototypage d'un système construit sur un environnement Prolog. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1989. Français. NNT : . tel-00306505

HAL Id: tel-00306505
https://theses.hal.science/tel-00306505

Submitted on 28 Jul 2008

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

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

THESE

présentée à

L' Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR de 3ème CYCLE
(informatique)

par

Alain LUCCI

Production de logiciels pour l'enseignement: une expérience de prototypage d'un système construit sur un environnement Prolog

Thèse soutenue le 17 février 1989, devant la commission d'examen.

J. MOSSIERE	Président
M. QUERE	
J.P. PEYRIN	Examinateurs
M. MILCHBERG	
P.C. SCHOLL	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales
(Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme
d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agréées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G

BOREL Joseph

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

ROWE Alain

E.N.S.I.M.A.G

COURTIN Jacques

E.F.P.

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

A Véronique et à Florian

Je tiens à remercier,

Monsieur Jacques Mossière, Professeur à l'I.N.P.G., Directeur de l'E.N.S.I.M.A.G., pour l'honneur qu'il me fait de présider le jury de soutenance. Son dynamisme et ses grandes qualités humaines sont pour moi un exemple,

Madame Maryse Quéré, Professeur à l'Université de Nancy 2, d'avoir accepté de participer à ce jury. Son accueil et son soutien amical et bienveillant, lors des différentes conférences où je présentais des étapes de ma recherche, m'ont beaucoup touché,

Monsieur Jean-Pierre Peyrin, Maître de conférences, à l'Université Joseph Fourier, responsable de la MST ESI, qui fut un de mes enseignants à l'Institut de Programmation de Grenoble, et qui a su me communiquer sa passion pour l'enseignement,

Monsieur Mauricio Milchberg, Responsable de Formation de Bull à Paris, qui a dirigé ce travail. Qu'il trouve ici toute ma reconnaissance pour le temps important qu'il m'a consacré, et pour ses efforts patients d'amélioration et d'approfondissement de cette recherche. Qu'il soit certain de l'amitié que je lui témoigne,

Monsieur Pierre-Claude Scholl, Professeur à l'Université Joseph Fourier, Directeur de l'U.F.R. Informatique et Mathématiques Appliquées, qui a suivi mes travaux de recherche, depuis leur début. Je le remercie particulièrement de m'avoir accueilli dans l'équipe Méthodologie de la Programmation et d'avoir accepté la responsabilité de ces recherches.

Je voudrais terminer, en dédiant cette thèse à mon épouse et à mon fils, qui ont été ma principale motivation. Sans eux, ce travail ne serait pas.

SOMMAIRE

Introduction 1

1. La production de logiciels pour l'enseignement 7

1.1. Les logiciels pour l'enseignement 7

1.1.1. Les tutoriels 8

1.1.2. Les logiciels informatifs 10

1.1.3. Les logiciels de simulation 11

1.1.4. Les environnements d'apprentissage 12

1.1.5. Les tuteurs intelligents 12

1.2. Approche classique de la production de didacticiels 15

1.2.1. Modèle de cycle de vie d'un didacticiel 15

1.2.2. Intervenants et structures de production 20

1.2.3. Outils de production 21

1.3. Prototypage et développement incrémental 26

1.3.1. Le prototypage en génie logiciel 27

1.3.2. Le développement de systèmes experts 29

1.4. Conclusion 31

2. Eaolog: un premier niveau de fonctions 33

2.1. Fonctionnalités auteur 33

2.1.1. Moyens d'expression fournis à l'auteur 33

2.1.2. Environnement d'édition 37

2.2. Un exemple de dialogue 38

2.2.1. Description d'un langage de réponse 40

2.2.2. Description des objets d'interface et du dialogue 45

2.3. Prolog et la production de logiciels pour l'enseignement 48

2.3.1. Réalisation des primitives EAO: approche procédurale 50

2.3.2. Réalisation des éditeurs: approche relationnelle 54

2.3.3. Apports de l'approche grammaire 56

2.4. Conclusion 61

3 Analyse pédagogique du logiciel Idalgo 63

3.1. Contexte de l'expérimentation 64

3.1.1. Exemples de logiciels pour l'enseignement de la programmation 64

3.1.2. Contexte pédagogique du logiciel Idalgo 69

3.2. Cahier des charges pédagogiques du logiciel Idalgo 73

3.2.1. Objectifs pédagogiques 73

3.2.2. Activités pédagogiques 74

3.2.3. Modèles de dialogues des activités 75

3.2.4. Variations dans la présentation des objets 87

3.3. Conclusion 94

4 Production du logiciel Idalgo: le système Tangram 97

4.1. L'approche de production 98

4.1.1. Analyse des besoins des différents intervenants 99

4.1.2. Fonctionnalités et architecture du système de production 100

4.1.3. Expression des connaissances 102

4.2. Le système Tangram 107

4.2.1. Connaissances sur la notation algorithmique 107

4.2.2. Modèles de dialogues des activités 117

4.2.3 Schémas, instances et algorithmes 122

4.3. Le logiciel Idalgo 131

4.3.1. Fonctionnalités de l'environnement apprenant 131

4.3.2. Description des outils de manipulation d'algorithmes 132

4.3.3. Critiques et extensions d'Idalgo 134

Conclusion 139

Annexe: Le traitement séquentiel 143

Classement par thèmes des références citées 151

Références bibliographiques 153

INTRODUCTION

Le travail présenté dans cette thèse concerne la production de logiciels pour l'enseignement. Il fait partie d'une série de travaux conduits au sein du Laboratoire de Génie Informatique depuis 1981 dans le domaine de l'enseignement assisté par ordinateur et plus particulièrement sur la question du génie didacticiel.

L'objectif général est d'affiner les méthodes et les outils mis à la disposition des enseignants pour utiliser efficacement l'ordinateur dans l'enseignement. La problèmatique de génie logiciel doit tenir compte des spécificités de l'enseignement assisté par ordinateur:
- multiplicité des intervenants de compétences diverses,
- difficulté de modélisation de l'apprentissage,
- difficulté de spécification des logiciels,
- nécessité d'un grand paramétrage pour que le logiciel puisse être adapté à différents contextes d'enseignement,
- forte évolutivité du logiciel lors de son développement, notamment du fait de la validation pédagogique,
- forte distance qualitative entre les logiciels existants sur le marché et les potentialités de l'informatique.

Un premier projet, le projet Mosaïque ([Ada 83], [AbS 84], [Sch 83]) développé entre 1981 et1983 a permis, autour d'une expérience concrète de réalisation d'un didacticiel en vraie grandeur d'expérimenter et d'évaluer une approche de production fondée sur une interaction permanente entre

enseignants et informaticiens. L'hypothèse était de libérer au maximum les enseignants du souci d'une compétence en informatique.

Deux didacticiels ont été élaborés, correspondant chacun à une trentaine d'heures pour l'apprenant. Il s'agissait d'un enseignement de Français-Langue Etrangère, conçu comme soutien à un cours principal, et s'adressant à des arabophones faux débutants dans l'apprentissage du français. Les principes généraux, linguistiques et pédagogiques qui ont guidé la conception des deux didacticiels étaient les suivants ([Sch83], [AbS84]):

- centrer l'étude sur la langue et sur l'apprenant,
- organiser le travail de l'apprenant selon différentes approches: réflexion, reconnaissance, production,
- avoir une stratégie où l'erreur sert de base de travail et n'est pas pénalisée.

Du point de vue des outils, on ne disposait à l'époque d'aucun système de production, et les informaticiens du projet ont réalisé un noyau de système adapté à la production envisagée tout en évaluant les besoins pour un système plus général. L'expérience était d'autant moins facile que le didacticiel incluait une communication multi média (diapositives, sons, textes) et que les matériels disponibles à ce sujet étaient rudimentaires.

La recherche s'est ensuite poursuivie par diverses expérimentations sur les outils de production disponibles ou envisageables, et par une réflexion plus novatrice sur la nature des activités pédagogiques adaptées à une utilisation de l'ordinateur. Un nouveau champ d'application a alors été privilégié, à savoir l'enseignement de l'algorithmique et de la programmation. Ceci a débouché sur des propositions concernant la structure pédagogique d'un environnement d'EAO de la programmation ([LPS83], [Sch84], [PeG85], [AGP85], [ScP88]).

A l'heure actuelle ces travaux sont concrétisés par un prototype de système de production (Tangram) fondé sur un environnement Prolog (que nous présentons dans cette thèse) et par les réalisations du projet Arcade ([CGL88]), plus particulièrement dédié à l'étude de nouvelles formes d'activités pédagogiques utilisant l'ordinateur et à la conception d'un laboratoire informatisé permettant d'intégrer ces activités.

Travaux réalisés

J'ai débuté mes travaux de recherche dans le contexte du projet Mosaïque. Je me suis intéressé, au cours de mon DEA, à l'utilisation de Prolog comme moyen de spécification des didacticiels par les enseignants: l'hypothèse était que de par leur propre culture, les enseignants étaient à même de s'approprier facilement le formalisme des grammaires de Prolog. Ceci a été expérimenté avec des enseignants de Français Langue étrangère ([Luc83a]). Ce travail, présenté en 1984 à la première conférence francophone sur l'EAO ([Luc84]), contenait les premiers éléments du système de production **Eaolog** que j'ai développé par la suite.

En parallèle, j'ai réalisé une nouvelle version du système de production Mosaïque, le système **Voyelles** ([Luc83b]) utilisée par d'autres chercheurs ([Zam84], [Lie85]), et dans des formations à l'EAO que j'ai assurées. J'ai par ailleurs avancé ma réflexion sur l'expressivité de Prolog en présentant ce moyen d'expression à des publics divers ([Luc86c]).

J'ai développé le système **Eaolog** en Prolog, successivement sous les environnements LSI-Prolog sous RT11 ([CMF80]) puis RSX 11M sur Plessey 23 VZJ, puis C-Prolog ([PPW84]) sous Unix sur Micromega, et enfin Prolog II version 2.2 ([PRO85], [PRO85b]) sur Macintosh. La version

actuelle du système fonctionne, avec l'interpréteur Prolog II version 2.4 ([PRO 87]), sur Macintosh Plus, Macintosh SE et Macintosh II.

J'ai ensuite poursuivi mon travail dans le contexte de l'enseignement de la programmation. A partir des problèmes rencontrés lors de la réalisation de maquettes dans ce domaine ([Zam84], [Agu85], [Lie85], [Gue85]) à l'aide de différents systèmes de production (Mosaïque et Voyelles déjà cités, Diane-Arlequin [Cas84], Diane-Editeur fonctionnel [Eur85]), j'ai défini une nouvelle couche logicielle pour le traitement assisté de notations, spécifiées par des grammaires, le système **Tangram** et un logiciel pour une initiation à une démarche algorithmique, le logiciel **Idalgo**, centré sur l'utilisation de schémas d'algorithmes pour l'écriture de programmes itératifs ([Sch79], [ScP88]).

Lors de la conférence EAO87, j'ai ainsi présenté des résultats autour de Tangram, notamment en ce qui concerne la paramétrisation des logiciels pour l'enseignement, du double point de vue de leur appropriation par les enseignants et des conséquences sur les environnements de production.

Présentation de la thèse

Le texte présente des réflexions sur deux thèmes:
- expressivité des moyens de communication offerts pour la conception et la production d'un logiciel pour l'enseignement,
- étude d'une approche de production.

En ce qui concerne le premier point, l'idée est de choisir un langage de spécifications exécutable accessible à la fois aux enseignants, auteurs ou utilisateurs, et aux informaticiens. L'approche de production est fondée sur les éléments suivants:

- application d'un modèle de cycle de vie avec prototypage et développement incrémental,
- conception d'un logiciel pour l'enseignement en termes d'activités pédagogiques: chaque activité est justifiée par un objectif pédagogique global et conduit à la spécification d'un modèle de dialogue. Un tel modèle est ensuite instantié pour produire un dialogue particulier.
- mise en évidence de modèles d'objets représentant la connaissance du domaine enseigné: ces modèles sont utilisés aussi bien par les enseignants au cours de la production que par les apprenants au cours de l'exploitation.
- description des dialogues EAO en termes de modèles d'objets EAO et des actions associées.
- proposition d'une architecture comportant plusieurs niveaux: un environnement général de programmation, une première couche de fonctions générales EAO, une deuxième couche de fonctions dédiées à un thème d'enseignement et enfin, les logiciels particuliers pour un enseignement dans ce thème.

Le travail s'appuie sur deux expérimentations, en enseignement du Français Langue Etrangère et pour une initiation à l'algorithmique.

Il est concrétisé par divers logiciels conçus et réalisés à partir d'un environnement Prolog: Eaolog, Tangram et Idalgo. **Eaolog** correspond à la première couche logicielle définie ci-dessus. **Tangram** construit avec Eaolog, offre des fonctions spécifiques à l'enseignement de l'algorithmique: l'objet fondamental du domaine enseigné est la notion d'algorithme. Tangram fournit la possibilité de définir une notation algorithmique et de manipuler des algorithmes que ce soit au niveau de la production d'un logiciel pour l'enseignement ou au niveau de son exploitation par un apprenant. **Idalgo** est un exemple de logiciel construit à partir de Tangram: quatre activités pédagogiques sont définies (construction en mode

lecture ou écriture, traduction, exécution) et donnent lieu à autant de modèles de dialogues. L'ensemble est intégré de manière à répondre aux besoins d'un informaticien, d'un enseignant auteur, d'un enseignant utilisateur et d'un apprenant.

La thèse est articulée en quatre chapitres. Le premier chapitre pose le problème de la production de logiciels pour l'enseignement en référence à une classification de ces logiciels, aux outils existants, aux résultats acquis en génie logiciel et aux apports potentiels du prototypage et d'une approche bases de connaissances.

Le deuxième chapitre présente Eaolog, qui offre des fonctions générales de conception des dialogues d'un logiciel d'enseignement. L'expressivité du langage Prolog y est étudiée, tant pour la spécification par les auteurs, que pour l'implémentation par les réalisateurs.

Le troisième chapitre décrit le cahier des charges pédagogiques du logiciel Idalgo, en le situant par rapport à des expériences sur l'enseignement de la programmation. Ceci illustre ainsi les premières étapes de l'approche de production étudiée.

Le quatrième chapitre décrit l'approche de production par une présentation des fonctionnalités et de l'architecture du système Tangram. Ceci est illustré à propos de la production du logiciel Idalgo.

Chapitre I

LA PRODUCTION DE LOGICIELS POUR L'ENSEIGNEMENT

Ce chapitre pose le problème de la production de logiciels pour l'enseignement. La première partie (§1.1) est consacrée à une classification des logiciels pour l'enseignement: tutoriels, logiciels informatifs, logiciels de simulation, environnements d'apprentissages et tuteurs intelligents.

Une deuxième partie analyse l'approche classique de la production de logiciels d'enseignement (§1.2): cycle de vie, intervenants, structures et outils de production. Les problèmes posés par cette approche conduisent à étudier (§1.3) des modèles de cycle de vie avec prototypage et développement incrémental, issus des travaux en génie logiciel et en intelligence artificielle pour le développement de systèmes experts.

Nous concluons sur les apports potentiels de ces approches dans la production de logiciels pour l'enseignement.

1.1. Les logiciels pour l'enseignement

L'Enseignement Assisté par Ordinateur (E.A.O.) est le terme le plus couramment utilisé en France pour désigner l'utilisation de l'informatique dans l'enseignement. Cette dénomination est souvent jugée peu satisfaisante ([PiB87]), ce sigle unique ne pouvant recouvrir l'ensemble des applications. La terminologie anglo-saxonne, plus précise, associe un terme à chaque champ d'application:

- Computer Assisted Instruction (C.A.I.) correspond à notre sigle E.A.O. Ceci concerne l'utilisation de l'ordinateur à des fins de transmission de connaissances,

- Computer Assisted Learning (C.A.L.) désigne l'ordinateur utilisé comme une source d'apprentissage,
- Computer Managed Instruction (C.M.I.) ou enseignement géré par ordinateur, le mot gestion étant pris essentiellement au sens de suivi et de l'évaluation de l'apprenant.

"Cette difficulté de terminologie s'est transposée sur celle des logiciels. Différents noms ont été employés ces dernières années: didacticiels, tutoriels, imagiciels, ludiciels..., aucun ne prétend être un terme générique, ils reflètent simplement un choix pédagogique du concepteur" ([PiB87]). Pour notre part, nous parlerons de *logiciels pour l'enseignement*

Schématiquement on peut distinguer différents types de logiciels pour l'enseignement: les tutoriels, les logiciels informatifs, les logiciels de simulation, les environnements d'apprentissage et les tuteurs intelligents. En fait un logiciel pour l'enseignement chevauche plusieurs des types cités. Nous présentons ici chacun de ces types de logiciels.

1.1.1. Les tutoriels

Dans ce type de logiciel, l'enseignant souhaite présenter à l'apprenant un sujet avec une stratégie pédagogique directive. Un tutoriel présente donc des connaissances et vérifie que les objectifs pédagogiques sont atteints en demandant à l'apprenant de répondre à des questions.

L'Agence de l'Informatique (ADI) a réuni en 1981, un groupe de travail pour définir les spécifications des outils nécessaires aux développements de logiciels pour l'enseignement. Sans parler pour l'instant des outils envisagés dans le rapport issu de ces travaux ([ADI81]) nous en présentons ici la terminologie que nous utiliserons par la suite.

Un *didacticiel tutoriel* est un ensemble structuré de *dialogues*. Les liaisons entre dialogues sont décrites dans la structure du didacticiel par des

unités de liaison. A chaque dialogue est associée une unité de liaison qui décrit les conditions de passage d'un dialogue à l'autre .

Un dialogue est un ensemble structuré de *situations pédagogiques* entre lesquelles existent des liaisons. On retrouve donc ici la même structure que pour les didacticiels. A chaque situation pédagogique est attachée une unité de décision permettant d'enchaîner les situations pédagogiques présentées à l'apprenant .

Une situation pédagogique regroupe une *unité d'interaction* et une *unité de décision*. L'unité d'interaction est composée d'actions prises parmi les suivantes:
- *envoi d'informations* à l'apprenant,
- *envoi d'une sollicitation*,
- *réception d'informations* de l'apprenant,
- *analyse de réponses*,
- *gestion d'informations* pédagogiques.

L'*envoi d'informations* englobe toutes les actions, élémentaires ou composées, de présentation d'informations à l'apprenant, sous forme de textes, de dessins, d'images fixes ou animées ou de messages sonores.

L'*envoi d'une sollicitation* demande à l'apprenant de réagir aux informations reçues.

La *réception d'informations* de l'apprenant englobe toutes les actions élémentaires ou composées de réception de messages qui peuvent être de sources différentes (clavier, crayon lumineux, tablette graphique, source sonore).

L'*analyse de réponse* consiste à identifier s'il y a coincidence entre la réponse de l'apprenant et l'une des réponses prévues par l'auteur. Elle détermine la correction de la réponse et/ou la nature de l'erreur détectée.

La *gestion d'informations pédagogiques* est utilisée pour la mise à jour de l'environnement du dialogue et pour permettre la mise en oeuvre des prises de décision pédagogique (unités de décisions ou unités de liaisons).

Il faut remarquer que l'analyse de réponse peut être très simple dans le cas où la réponse se réduit à un caractère, une touche de fonction ou à la désignation d'une zone de l'écran. Par contre elle peut être plus complexe dans le cas d'une phrase, d'une formule ou d'un message graphique.

Comme l'indiquent D. Coulon et D. Kayser dans [CoK79], plus fine sera l'analyse des expressions émises par l'apprenant, meilleure sera l'adaptation de l'assistance pédagogique à son cas. On peut, dans ce sens, définir une hiérarchie (du plus contraignant au plus ouvert) des messages à analyser. Par exemple, dans le cas des messages écrits, on peut distinguer les niveaux suivants:
- reconnaissance d'un caractère: ce niveau correspond aux Questions à Choix Multiples (Q.C.M.).
- analyse lexicale: si la réponse se compose de plusieurs mots, cette analyse permet de les délimiter. On peut donc faire une comparaison avec une liste de mots attendus.
- analyse syntaxique: cette analyse permet de déterminer si une phrase donnée appartient à un ensemble de phrases décrit par une grammaire.
- analyse syntaxique et vérification sémantique: cette analyse vérifie qu'une phrase ou formule est correcte syntaxiquement et sémantiquement.

1.1.2. Les logiciels informatifs

"Ce type de logiciel sert à proposer et présenter des connaissances, dans un domaine donné, sans chercher à vérifier leur acquisition par l'apprenant. Il s'agit donc de mini-encyclopédies électroniques, apparentées au fond aux banques de données"([For 85]).

L'accés aux informations stockées, peut-être séquentiel (logiciel "tourne-pages"), par menus (dictionnaire feuilletable), par requêtes (questions ouvertes). Dans ce dernier cas, l'enseignant doit définir un langage d'interrogation. On retrouve ici le même problème que pour l'analyse de réponses des tutoriels, sous la forme d'analyse de requêtes.

"Il est probable que l'évolution technologique (apparition des CD-ROM de très haute capacité) ouvrira un avenir nouveau et beaucoup plus ambitieux à ces didacticiels informatifs en rendant possibles de véritables banque de données personnelles, exploitables soit directement soit au travers d'un filtre logiciel didactique" ([For85]).

Une autre tendance de ce type de logiciels est liée à l'apparition de systèmes d'hypertexte ou d'hypermédia. "Un système d'hypertexte emploie les possibilités électroniques pour rechercher les passages intéressants d'un document et il utilise des écrans graphiques à haute résolution pour les visualiser" ([Sav87]). Ces systèmes favorisent une exploration non séquentielle des documents, des graphiques statiques ou dynamiques , du son ou des images pouvant enrichir le texte. A titre d'exemple de cette tendance, on peut citer: le livre electronique EBOOK3 ([Sav87]), et les logiciels informatifs réalisés sous Hypercard sur Macintosh.

1.1.3. Les logiciels de simulation

Dans ce type de logiciel l'apprenant expérimente de façon dynamique un phénomène modélisé. Construits autour de modèles paramétrés, ces logiciels simulent les réactions des systèmes. Le but est d'amener l'apprenant à observer le phénomène réel simulé. Différentes expériences ont été réalisées en sciences physiques, en biologie, en gestion (jeux d'entreprises) etc.

Dans [Jer81], deux voies opposées et complémentaires sont présentées pour l'utilisation de simulations en EAO:
- la première correspond à l'analyse d'un phénomène naturel avec des simulations comportementales ayant un objectif soit méthodologique, soit dynamique, ou encore opérationnel. Les apprenants connaissent, dans ce cas, le modèle sous-jacent et utilisent les simulations pour vérifier l'adéquation du modèle ou pour résoudre des problèmes,

- la deuxième voie offerte est celle d'un entraînement de l'esprit des enseignés au raisonnement inductif par l'utilisation de simulations modélisantes. Les apprenants cherchent à découvrir le modèle sous-jacent en réalisant une expérimentation fictive.

1.1.4. Les environnements d'apprentissage

Dans les logiciels de ce type l'accent est mis sur la découverte et l'exploration plutôt que sur la transmission des connaissances. L'approche Logo de Seymour Papert ([Pap73], [Pap81]) issue du courant de pensée piagétien (apprentissage spontané en interaction avec l'environnement) est l'exemple le plus célébre de cette tendance. Logo a permis l'apparition de logiciels appelés généralement "univers Logo", qui cherchent à élargir le champ d'application du Logo initial (celui de la géométrie), pour aborder d'autres domaines tels que la langue écrite, la musique, etc.

1.1.5. Les tuteurs intelligents

"L'E.I.A.O est une discipline issue de l'E.A.O. et de l'I.A. (Intelligence Artificielle). Le sigle signifie Enseignement Intelligement Assisté par Ordinateur, mais gagne à être traduit par Enseignement, Intelligence Artificielle et Ordinateur, pour ne pas suggérer qu'en dehors de l'E.I.A.O, les produits de l'E.A.O sont tous stupides"([Nic86]).

L'idée générale est d'insérer les connaissances d'un expert dans un environnement logiciel capable d'exploiter ces connaissances. "On suppose que l'ordinateur puisse se mettre dans la même situation que l'apprenant (ignorance de la solution, capacité d'en construire une à partir des données connues de l'apprenant), mais aussi qu'il connaisse les principales erreurs que les apprenants ont l'habitude de commettre, afin de donner une interprétation sensée à d'éventuelles réactions inappropriées" ([Cok79]).

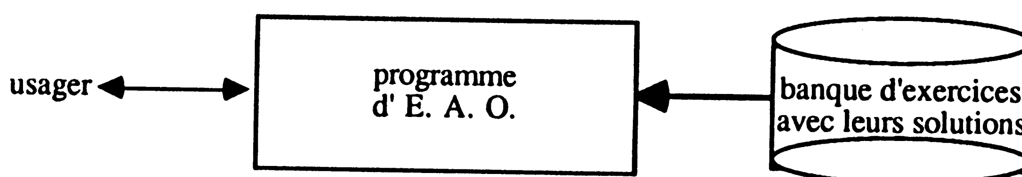
Le premier produit de type E.I.A.O. remonte aux années 1970: Scholar ([Car70]) qui enseignait la géographie de l'Amérique du sud, utilisait une représentation des connaissances lui permettant de ne pas avoir une réponse préenregistrée associée à chaque question.

Le logiciel Guidon ([Cla79], [Cla83]) a pour objet l'apprentissage du diagnostic de pathologies sanguines en médecine. Lié au système expert Mycin ([Sho76]) , c'est un exemple de la transition entre un système expert destiné à la médecine et un système expert à buts pédagogiques.

De nombreux exemples de tuteurs intelligents dans différents domaines (médecine [FFB84], physique [Pal86], mathématiques [Nic87], langues [Hat85], [Sch87], informatique [Cou87], [BBF87], ...) sont présentés dans [SlB82] [Cla86] [Niv87] et [Wen87].

La tendance est à séparer l'expertise en résolution de problèmes, et les stratégies pédagogiques. Dans [Lel87], R Lelouche présente cette évolution de l'E.A.O. traditionnel, à l'E.I.A.O. expert du domaine et à l'E.I.A.O. psycho-pédagogue (figures 1.1, 1.2 et 1.3).

figure 1.1: structure d'un programme d'E.A.O. traditionnel



Un système d'E.I.A.O. expert du domaine, dispose d'une banque de connaissances, lui permettant de résoudre lui-même les problèmes qu'il pose. Il n'a donc plus besoin que la solution des exercices soit dans la banque. "On peut même imaginer de remplacer le sélecteur d'exercices par un générateur d'exercices; la banque d'exercices devient alors inutile" [Lel87].

figure 1.2: structure d'un système d'E.I.A.O. expert du domaine

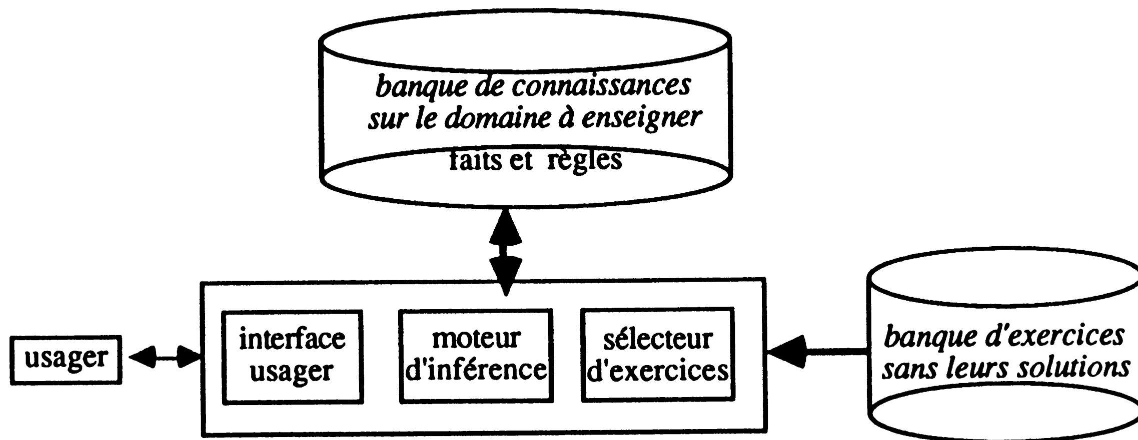
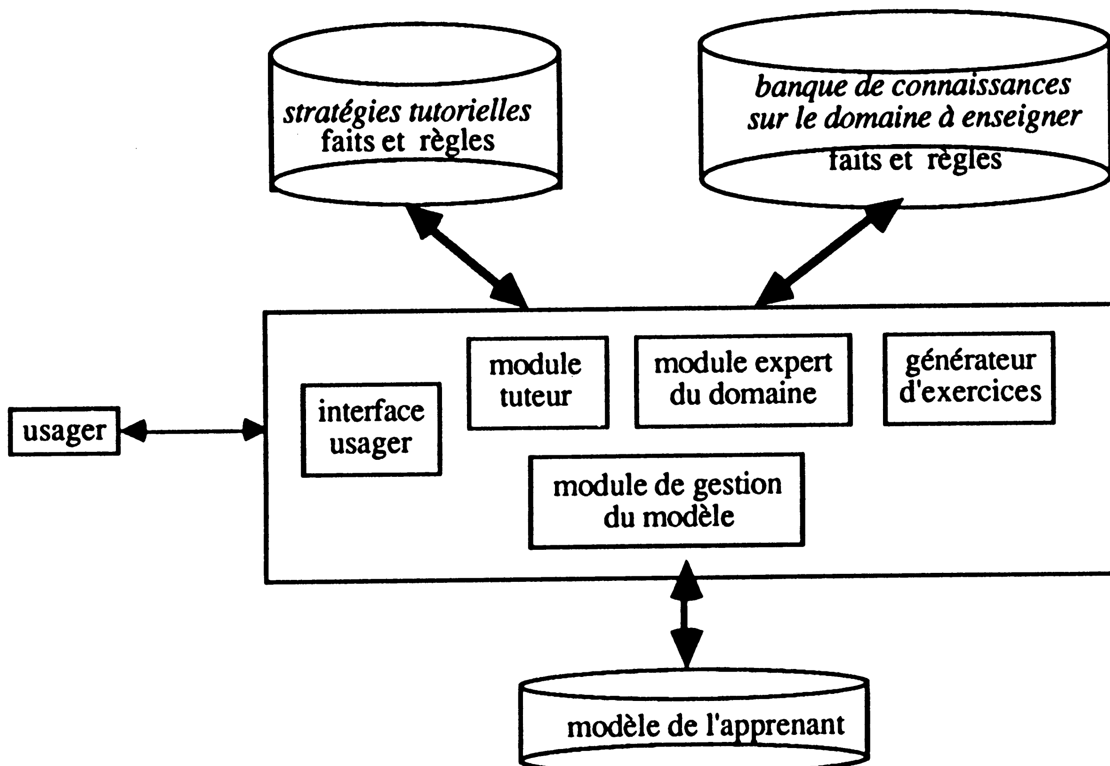


figure 1.3: structure d'un système d'E.I.A.O. psycho-pédagogue



Un système d'E.I.A.O. psycho-pédagogue, comporte de plus,

- une base de connaissances sur les stratégies tutorielles: les faits en sont les différents types d'intervention du logiciel: répéter, reformuler,

corriger, expliquer, approuver, questionner, etc.; les règles indiquent les modalités de ces interventions: quand, comment, pourquoi, ... intervenir ?
- un modèle de l'apprenant comprenant des connaissances générales et des connaissances liées au domaine: comment apprend-t-il ?, comment découvre-t-il (le mieux) de nouveaux concepts ?, ce que l'apprenant sait et ce qu'il sait faire, etc.

Remarque: on retrouve l'idée de modélisation d'un utilisateur dans d'autres contextes informatiques (environnements de programmation, recherche documentaire, etc...). Les représentations employées y sont très souvent très pauvres: l'utilisateur est codé par exemple par une variable pouvant prendre trois valeurs: expert, normal, novice.

1.2. Approche classique de la production de didacticiels

Par analogie au génie logiciel, on parle de génie didacticiel pour l'ensemble des méthodes, techniques et outils de production de didacticiels de qualité industrielle. Le génie logiciel a mis en évidence différents *modèles de cycle de vie* d'un logiciel (cycle en cascade, incrémental,...) et fournit des *méthodes et des outils* pour les différentes phases liées aux différents types d'*intervenants* du processus de production: client (utilisateur ou non), spécifieur, réalisateur, utilisateur. Nous présentons ici l'approche classique de la production de didacticiels en ces termes: modèle de cycle de vie (§1.2.1), intervenants et structures de production (§1.2.2), outils de production (§1.2.3).

1.2.1. Modèle de cycle de vie d'un didacticiel

On distingue habituellement dans le cycle de vie d'un didacticiel les différentes étapes suivantes ([Rec85], [Fis85], [HBB86]): l'étude d'opportunité, l'analyse pédagogique, l'élaboration du scénario, la

réalisation informatique, la validation pédagogique et l'expérimentation, l'industrialisation et la diffusion, l'installation et l'exploitation.

a) étude d'opportunité

Cette étape doit mettre en évidence l'opportunité du projet et définir son contexte de réalisation et sa faisabilité. Elle conduit à la production d'un document permettant de juger de l'intéret pédagogique du projet vis à vis d'autres moyens pédagogiques utilisables et de son caractère réaliste.

Le dossier d'opportunité produit lors de cette étape doit définir le sujet traité (domaine et thèmes), les caractéristiques de la population cible, les objectifs pédagogiques poursuivis, le rôle du logiciel envisagé et sa situation par rapport à des logiciels existants et finalement le cadre de réalisation et de mise en oeuvre.

b) analyse pédagogique

Le point de départ de cette étape est un dossier d'opportunité considéré comme réaliste. Cette étape s'appuie sur deux thèmes de travail:
- définition des objectifs: on décompose les objectifs terminaux en objectifs intermédiaires et prérequis, on recherche des activités d'apprentissage en liaison avec les objectifs définis, et l'on caractérise diverses situations d'apprentissage.
- analyse de la matière: il s'agit d'une analyse détaillée du contenu en fonction de la population cible et d'une mise en forme de la matière à enseigner.

Cette étape conduit à la production d'un cahier des charges pédagogiques qui définit le sujet traité, la population cible, le niveau d'entrée (prérequis), les objectifs terminaux et intermédiaires, les activités d'apprentissage, et les situations d'apprentissage.

c) élaboration du scénario du logiciel

"Le scénario est la maquette sur papier de ce que sera le logiciel. Le

scénario reproduit dans le détail, mais sur le papier, les interactions entre l'élève et l'ordinateur" ([Rec85]). Il décrit:

- la décomposition du didacticiel en dialogues: chaque dialogue est défini par son objectif pédagogique et l'activité demandée à l'apprenant,
- les cheminements possibles de l'apprenant dans le didacticiel: enchaînements de dialogues qui marquent de grandes étapes de progression dans le didacticiel,
- la décomposition des dialogues en unités d'interaction: présentations de scènes et de sollicitations, réception d'information, analyse de réponses, réactions aux reponses après analyse, "suivi" de l'activité de l'apprenant, requêtes disponibles.

Dans [For85] il est mis en évidence que cette notion de scénario renferme en elle-même une complexité qui est loin d'être apparente à tous. "Une 'explication' de leçon ou d'exercice qui, faite de maître à maître, se réduirait à un échange de quelques phrases, va, sous la forme d'un scénario de didacticiel, prendre une ampleur insoupçonnée et exiger une précision dans les détails décourageante; il faudra tout dire, et en particulier: donner le texte complet de toutes les questions, réponses et explications; préciser l'emplacement où ils doivent apparaître dans l'écran; dire quel chemin emprunter en cas de telle ou telle réponse donnée par l'apprenant; prévoir ces réponses et aussi toutes les autres; dire quand et quelle partie de l'écran effacer etc".

d) réalisation informatique

Le passage du scénario à un programme informatique est effectué soit à l'aide d'outils spécialisés, langages ou systèmes d'EAO (cf § 2.3), soit en utilisant des outils de programmation non spécialisés en EAO. On retrouve ici les étapes centrales du modèle en cascade du cycle de vie d'un logiciel quelconque: conception générale, conception détaillée, codage, intégration et validation informatique (tests informatiques de bon fonctionnement).

e) validation pédagogique et expérimentation

La validation pédagogique est effectuée par l'enseignant-auteur par rapport à ce qu'il avait imaginé sur papier. Il peut vérifier entre autres si le cheminement dans le didacticiel correspond bien à la progression pédagogique souhaitée, si les analyses de réponses prennent bien en compte tous les cas prévus, si la présentation est bonne (bonne disposition des éléments sur l'écran et mise en relief des parties importantes), si le rythme convient , etc. Cette validation peut, bien évidemment, entraîner une remise en cause du scénario et donc de la réalisation.

Dans la phase d'expérimentation, le didacticiel est testé dans le contexte d'utilisation prévu. L'expérimentation permet de vérifier l'adéquation du didacticiel aux objectifs pédagogiques. Elle peut être effectuée dans un premier temps par les enseignants-auteurs, puis par d'autres enseignants n'ayant pas participé à la conception, et enfin et surtout avec des apprenants correspondant à la population cible du didacticiel. Ces expérimentations entraînent des modifications et des améliorations du logiciel, tenant compte des résultats des divers tests.

f) industrialisation et diffusion

Cette étape comporte entre autres la réalisation ou la mise en forme commerciale des documents pédagogiques et informatiques de présentation et d'accompagnement.

g) installation et exploitation

Un logiciel pour l'enseignement est en général exploité par des enseignants qui n'en sont pas les auteurs, enseignants utilisateurs). Médiateurs entre l'auteur et l'apprenant, ils interviennent à trois niveaux:

- adaptation du logiciel au contexte particulier d'enseignement, nous parlons d'appropriation du logiciel par l'enseignant utilisateur.
- animation de séances de travail des apprenants s'ils ont décidés d'être présents sur le site des séances d'EAO,

- exploitation des environnements didacticiels et élèves résultant de l'activation des didacticiels.

Soulignons l'importance de l'étape d'appropriation. "Par comparaison, nous pouvons prendre l'exemple du degré d'initiative de l'enseignant lorsqu'il construit un cours traditionnel. Chaque enseignant procède dans le cadre des orientations fixées pour une matière donnée au recueil, à la sélection et à l'adaptation des informations dont il estime avoir besoin. En quelque sorte, il souhaite "personnaliser" son produit" ([Mae84]).

En conséquence, la question de l'appropriation d'un didacticiel doit être prévue dès sa conception.

La figure 1.4. résume les étapes du cycle de vie que nous venons de présenter.

figure1.4: schéma général du cycle de vie d'un didacticiel

étapes	***documents***
étude d'opportunité	*dossier d'opportunité*
analyse pédagogique	*cahier des charges pédagogiques*
élaboration du scénario	*scénario en termes de dialogues et d'écrans*
réalisation informatique	*dossier de programmation*
expérimentation	*dossier d'expérimentation*
industrialisation et diffusion	*documentation*
installation et exploitation	*dossier d'utilisation*

1.2.2. Intervenants et structures de production

Au cours du cycle de vie d'un logiciel pour l'enseignement, on observe une grande diversité d'intervenants: l'apprenant qui utilise le logiciel d'enseignement, l'enseignant utilisateur ou responsable de formation qui en suit l'utilisation, l'enseignant auteur et l'informaticien. On trouve plusieurs structures de production ([Pat87]):

structure n°1 (exemple: Crédit Lyonnais)

étude d'opportunité	analyse pédagogique	élaboration du scénario	réalisation informatique

--------------------*enseignant auteur*------------------------------------->

Dans cette structure l'enseignant auteur est responsable de bout en bout de l'écriture du didacticiel. Il en assure la conception pédagogique et la réalisation informatique.

structure n°2 (exemple: Crédit Agricole)

étude d'opportunité	analyse pédagogique	élaboration du scénario	réalisation informatique

--------*enseignant auteur*------------------------->---*informaticien*-->

Dans cette structure en deux temps, le scénario conçu par l'auteur est repris par un informaticien qui assure la programmation. Cette structure de production avait été choisie dans le projet Mosaïque.

structure n°3 (exemple: CNAM)

étude d'opportunité	analyse pédagogique	élaboration du scénario	réalisation informatique

---------*professeur*------------->----*concepteur*->--*informaticien*-->

Le professeur analyse un sujet qu'il connaît bien et qu'il a l'habitude d'enseigner. Le concepteur élabore le scénario à partir du cahier des charges pédagogiques produit par le professeur. L'informaticien assure la programmation du scénario.

1.2.3. Outils de production

Parmi les outils logiciels à la disposition du réalisateur de logiciels d'enseignement on distingue habituellement des langages de programmation classiques, des langages *"plus"*, des langages auteurs et des systèmes d'EAO.

On remarquera qu'il y a eu en EAO la même évolution que dans l'ensemble de l'informatique: "le logiciel de base n'y était constitué au départ que du langage de programmation, puis on a vu apparaître les systèmes d'exploitation, les éditeurs de programmes, des environnements de programmation de plus en plus sophistiqués, pour aboutir au concept général de génie logiciel. En enseignement assisté par ordinateur, on est parti des langages auteurs, pour dériver sur la notion de système d'EAO, d'éditeurs et on parle maintenant de "génie didacticiel"" ([Que83]).

a) langages de programmation généraux

A l'origine beaucoup des logiciels d'enseignement ont été développés en utilisant des langages de programmation classiques non spécialisés (langage d'assemblage, Basic, Pascal, etc ...). En 1970, en France, J. Hebenstreit a créé un Langage Symbolique d'Enseignement (L.S.E.) qui a été à l'époque implémenté pour les ordinateurs de l'Education Nationale, dans le but d'enseigner l'informatique et a été utilisé pour uniformiser et rationaliser la production de logiciels d'enseignement.

On s'oriente actuellement vers l'utilisation d'autres langages, "C" par exemple, en profitant de l'environnement Unix. Ces langages exigent évidemment de bonnes compétences en informatique de la part du réalisateur du logiciel d'enseignement.

b) langages *"plus"*

Dans les utilisations de langages de programmation en EAO, "la "dérive" normale de l'outil est ce que l'on a parfois appelé un *langage "plus"* ([ADI81], [Que 83]), c'est-à-dire un langage augmenté d'une bibliothèque de procédures paramétrées réalisant les fonctions classiques de l'EAO: présentation, réception, gestion d'informations, analyse de réponse.

c) langages auteurs

Par comparaison aux langages de programmation, les langages auteurs ont été développés dans le but de faciliter la tâche du réalisateur, en lui proposant des moyens d'expression simplifiés et en offrant des facilités de programmation pour la gestion d'écran, pour la programmation des analyses de réponse, la manipulation et la définition de dialogues, etc.

Ces langages restent proches des langages de programmation classiques, en proposant un "habillage" d'un langage impératif et des bibliothèques de fonctions permettant la description des unités d'interaction et des dialogues (cf §1.1.1.).

On peut remarquer que les simplifications opérées dans la plupart de ces langages afin de faciliter leur apprentissage, se font au détriment de leurs possibilités expressives et méthodologiques: ils sont limités tant du point de vue des structures de données (entier, réel, logique et parfois tableaux) que des structures de contrôle (instruction conditionnelle et de branchement, et parfois instruction et/ou procédures la plupart du temps sans paramètres).

Par ailleurs, l'habillage des constructions algorithmiques classiques ne fait que cacher une réelle activité de programmation qui requiert toujours un minimum de formation sur les méthodes de programmation.

d) les systèmes d'EAO

Un système d'EAO est un ensemble de ressources matérielles et logicielles permettant la création, la modification et l'exécution de

didacticiels. Une caractéristique qui permet de différencier les différents systèmes d'EAO est l'existence ou non d'une forme langagière parmi les fonctionnalités présentes dans le système: les *systèmes langagiers* offrent aux enseignants un langage auteur; dans les *systèmes non langagiers* (systèmes guidés) ces fonctionnalités sont accessibles par exemple au travers d'éditeurs pour la définition et la manipulation des objets d'un didacticiel.

Les systèmes langagiers sont généralement plus puissants que les systèmes guidés et permettent de réaliser n'importe quel type de logiciels pour l'enseignement (tutoriel, simulation, etc), par contre ils sont plus difficiles à maîtriser pour un enseignant non informaticien.

Les systèmes guidés sont plus faciles d'emploi, ils ne nécessitent guère d'apprentissage, mais l'enseignant a beaucoup moins de liberté pour définir son logiciel: "il doit se couler dans le moule prévu par les concepteurs du système qu'il utilise; parfois tout un jeu d'options et de paramétrisations permet de pallier cet inconvénient; parfois les concepteurs du système ont résolu le problème en autorisant l'intégration dans le didacticiel d'unités préparées sous la forme de procédures écrites dans un langage de programmation" [MBP87].

Les systèmes d'EAO sont apparus très tôt sur les gros ordinateurs, nous pouvons citer par exemple: le système Plato réalisé par Control Data avec le concours d'universités ([Pla82]); le système Can 8 développé au Canada par Homecom Learning Systems Ltd et commercialisé en France par Bull; le système IMG d'IBM; le système OPE (Ordinateur Pour Etudiants). Le laboratoire OPE de l'Université Paris VII a été crée en 1966. Le système OPE, fonctionnant sur un ordinateur central (IBM 360/40), est la première expérience française dans le domaine ([Jac74]).

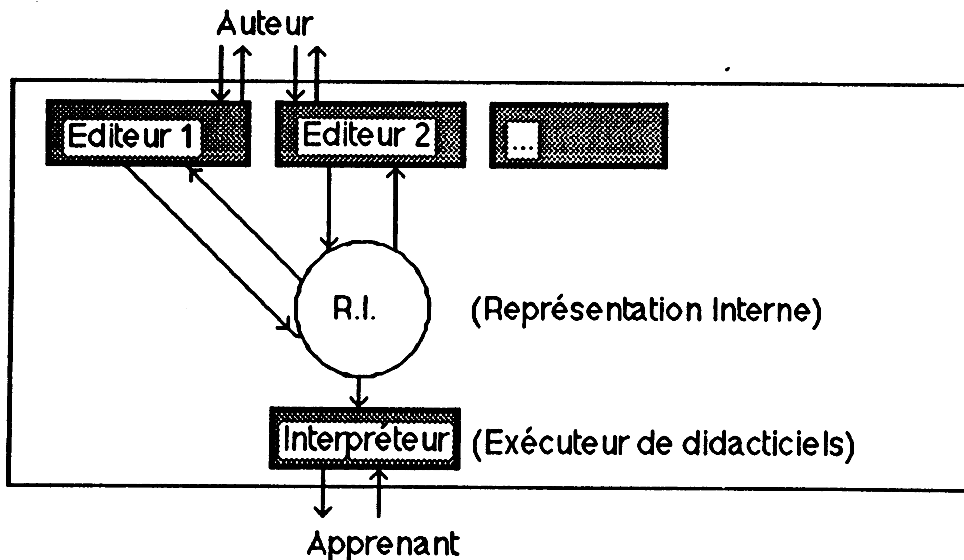
En France, en 1981, dans le cadre du Projet National EAO, l'Agence de l'Informatique a soutenu le developpement d'un système d'EAO: le système Diane ([ADI83]) qui constitue une tentative de normalisation

nationale en matière d'E.A.O. C'est un système d'aide au développement et à l'exécution de didacticiels conçu pour du matériel français de grande diffusion.

Un des apports du système Diane, inspiré des travaux d'outils d'aide au développement de logiciels en génie logiciel ([DKL83], [Hul83], [MRR82] [TeR81]), est d'avoir défini d'une manière formelle la représentation interne d'un didacticiel au moment de l'exécution. Une fois terminé, le didacticiel est stocké sous la forme d'une structure de données qui peut être interprétée (voir figure 1.5).

La représentation interne de Diane est destinée à codifier les didacticiels conçus par les auteurs au moyen des éditeurs. Un des interêts de cette approche est que plusieurs éditeurs adaptés à des modes d'expression différents peuvent être développés, l'outil de test, gestion et diffusion des didacticiels étant unique.

figure 1.5: le système Diane



A la fin du projet, quatre outils de production permettaient de générer de la R.I. Diane: l'éditeur fonctionnel; une forme langagière, l'éditeur Marion; un langage auteur, Arlequin; et un éditeur de simulation de cas.

e) exemples de systèmes et de langages auteurs

Il existe aujourd'hui de très nombreux systèmes et langages auteurs sur micro-ordinateurs. On trouvera dans [MBP85], [MBP87] et [Dia88] des études comparatives des principaux systèmes d'EAO disponibles: Diane-Editeur fonctionnel, Diane-Arlequin, Marion, Ego, Duo, Eva, Pen, Dr. Léo, Prof, Course Builder, etc. Nous détaillons trois de ces outils: Diane-Editeur fonctionnel, le système Dr. Léo et le système Prof.

Diane-Editeur fonctionnel est un exemple de système d'EAO sans langage auteur. Dans ce système ([Eur85]), didacticiels, dialogues et composants de dialogues sont des objets pédagogiques accessibles par des éditeurs. Ces objets: didacticiel, dialogue, situation, zonage, scène, texte, police, graphique, sollicitation, analyse, jugement, environnement, correspondent aux différentes notions mises en oeuvre lors de la création d'un didacticiel. Il y a autant d'éditeurs que de types d'objets. Chaque éditeur permet de créer, modifier et cataloguer les objets qui le concernent.

Le système Dr. Léo [Léo87] sur Macintosh est un exemple de système d'EAO avec langage auteur. Un didacticiel sous ce système est composé d'un "livre interactif", qui constitue le coeur du didacticiel et est écrit en langage auteur Dr. Léo, et de "livres annexes" (manuels, dictionnaires, etc...) qui sont des documents créés par les outils habituellement utilisés sur Macintosh (MacWrite, MacPaint, MacDraw, etc).

Le langage auteur Dr. Léo correspond à un "habillage" d'un langage impératif classique simple:

- structures de données: uniquement variables de types simples: entier, réel, chaine; pas de tableaux,
- structures de contrôle: "si alors sinon finsi", "cas fincas", "bloc finbloc" et instructions de branchements "reprendrebloc" et "finirbloc" permettant d'exprimer des structures de contrôle itératives,
- procédures: appel de procédure sans paramètres.

Le système Prof ([DSN87]) implémenté sur un Smaky 100 (équivalent à un Macintosh) est un exemple de système d'EAO avec langage auteur "graphique" (ou appelé générateur d'organigrammes). D'autres systèmes de ce type existent: le système Course Builder sur Macintosh ([CoB87]), le système de l'Irpeacs de Lyon ([Irp87]).

1.3. Prototypage et développement incrémental.

Pour définir une approche de production, nous partons de problèmes spécifiques à l'EAO, couramment admis par les spécialistes et que nous avons nous mêmes rencontrés dans nos diverses expériences rappelées en introduction.:
- diversité des intervenants du processus de production, avec des profils et des besoins différents.
- difficulté de la spécification du scénario par l'enseignant-auteur, en particulier pour l'expression de la structuration du didacticiel, du déroulement des dialogues et de l'analyse des réponses.
- évolutivité du logiciel durant son développement. Les deux étapes d'élaboration du scénario et de réalisation informatique sont fortement liées et sont soumises à de nombreux allers et retours, notamment du fait de la validation pédagogique et de l'expérimentation.
- manque de généralité des logiciels produits du point de vue leur appropriation par les enseignants utilisateurs pour leur exploitation dans un contexte d'enseignement particulier

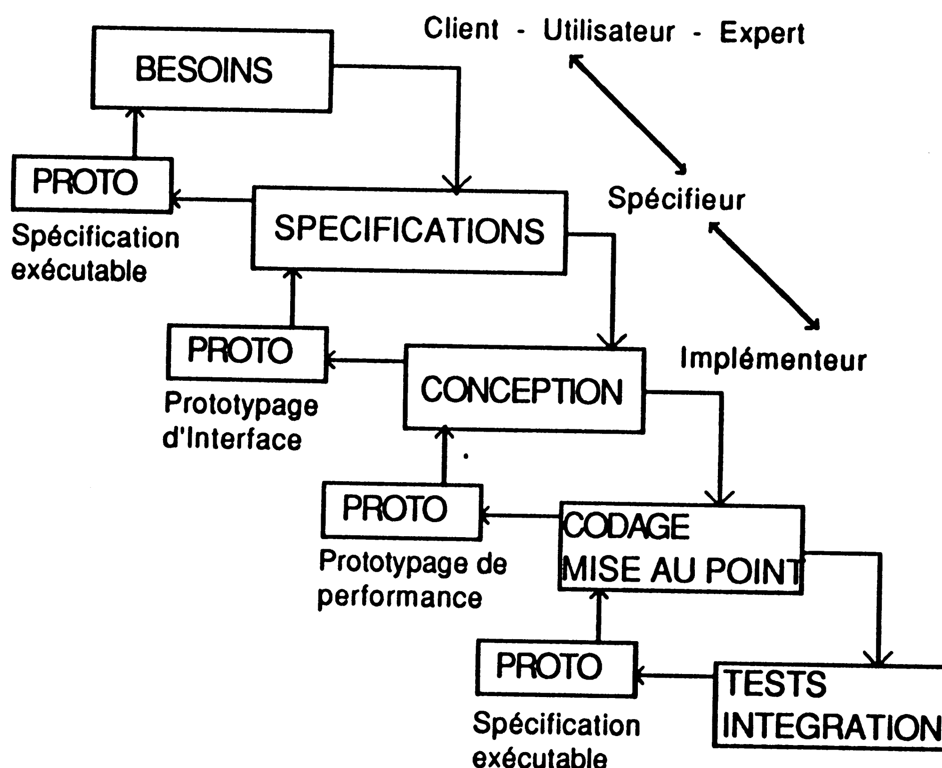
Ceci nous conduit à examiner les solutions proposées en génie logiciel et en intelligence artificielle, pour le développement de logiciels très évolutifs et /ou dont la spécification est difficile à établir. Nous nous centrons sur les modèles de cycle de vie avec prototypage (§ 1.3.1) et développement incrémental (§ 1.3.2).

1.3.1. Le prototypage en génie logiciel

Le prototypage facilite la communication entre les différentes personnes concernées par la réalisation et l'utilisation d'un logiciel. Par ailleurs, il permet d'appréhender plus facilement le comportement d'un système. Enfin, l'expérience acquise au cours de sa réalisation est précieuse pour développer le système final ([Cho85], [Cho88]).

De plus, selon l'étape du cycle de vie concernée, le prototype aura diverses utilités (figure 1.6): aider à préciser ou compléter la spécification, raffiner la conception, évaluer les performances, etc.

figure 1.6: Le prototypage dans le cycle de vie ([Cho88])



On distingue ainsi plusieurs types de prototypes: les *prototypes exploratoires* utilisés dans la phase de spécification des besoins, les *prototypes expérimentaux* lors de la conception globale et détaillée, les *prototypes évolutifs* pour le développement par versions successives. Cette

dernière approche consiste souvent à faire évoluer horizontalement (ajout de fonctionnalités) et verticalement (optimisation) un prototype exploratoire.

a) le prototypage exploratoire

La spécification des besoins est une des étapes critique dans le cycle de vie du logiciel. Très souvent, le client ne prend conscience de ses besoins qu'une fois le produit terminé et qu'il est confronté à la réalisation. Une motivation importante du prototypage exploratoire est donc la nécessité d'un outil de communication entre le client et le réalisateur.

Le prototype permet d'obtenir rapidement des réactions sur la conception du produit. Le client compare le comportement qu'il a réellement spécifié (ou voulait spécifier) et met en évidence les différences pour le réalisateur. Le prototype aide à clarifier les besoins du client, en permettant une étape de validation avant la réalisation. Il contribue à réduire dans la phase de spécification, les erreurs qui deviendraient très coûteuses en phase de réalisation.

On distingue habituellement deux types de prototypes exploratoires: les prototypes jetables et les prototypes évolutifs. Les prototypes jetables offrent comme nous venons de l'indiquer un outil de communication entre le client et le réalisateur et permettent une meilleure définition du cahier des charges du produit à réaliser. Les prototypes évolutifs correspondent à une première version du produit à réaliser.

b) outils du prototypage exploratoire

Ils s'appuient généralement, soit sur des langages de spécifications formelles exécutables, soit sur les langages issus des techniques d'intelligence artificielle: Lisp, Smalltalk ([Bez85]) et Prolog ([RLT85], [Leg87]) par exemple.

Dans [Leg87], consacré au prototypage de logiciels en Prolog, B. Legeard présente une vision des apports et des limites du langage Prolog

pour le prototypage. Les apports principaux en ce domaine sont les suivants:

- la déclarativité du langage qui permet un développement incrémental et rapide par ajout de nouvelles spécifications sous forme de faits ou de règles;
- l'indépendance par rapport aux problèmes de mise en oeuvre;
- et enfin, les structures récursives et le mécanisme de résolution qui entrainent une grande concision des programmes.

On est par contre limité par l'absence relative d'environnement de développement et l'absence d'outils graphiques adaptés au maquettage des interfaces homme/ordinateur, mais le domaine évolue rapidement.

Pour B. Legeard, l'intérêt majeur du langage Prolog en prototypage provient de sa déclarativité qui permet la convergence des approches par prototypage exploratoire et par spécifications formelles au travers du concept de spécifications exécutables.

1.3.2. Le développement de systèmes experts

On retrouve dans le développement d'un système expert l'intérêt du prototypage pour la communication entre l'expert et le cogniticien et du développement incrémental par enrichissements successifs.

a) les intervenants

Le développement d'un système expert met en jeu différents intervenants: expert du domaine, utilisateur, cogniticien, et informaticien. Le cogniticien aide l'expert du domaine à exprimer les connaissances du domaine d'expertise. On reconnaît ainsi une situation analogue à celle présentée plus haut pour l'EAO.

"La bonne démarche consiste à laisser à l'expert une grande liberté d'expression dans un premier temps, puis à lui suggérer des modalités adéquates de représentation des connaissances au fur et à mesure que leur nécessité se confirme" ([BHT86]).

b) les phases de développement

Le cycle de vie d'un système expert diffère notablement de celui d'un logiciel classique. Une méthode ([BHT86]) consiste à développer un système en trois phases:

- construction d'un prototype, capable de ne traiter qu'un sous-ensemble représentatif du problème total et ne possédant pas toutes les fonctionnalités d'un produit fini, notamment en ce qui concerne la communication avec l'utilisateur. Ce prototype permet de vérifier la faisabilité et de cerner les caractéristiques du produit final.
- amélioration du prototype: extension à l'ensemble du problème (augmentation de la base de connaissances et éventuellement des modes de raisonnement), validation de la base sur un ensemble significatif de cas, amélioration des interfaces, etc.
- implantation finale du système sur son site d'utilisation.

c) outils de développement

On peut schématiquement ordonner les outils disponibles en trois niveaux de complexité croissante, avec quelques variantes autour de cette classification ([Dem85]):

- les langages de programmation pour l'Intelligence Artificielle: langages fonctionnels (Lisp par exemple), langages de programmation en logique (Prolog par exemple), langages orientés objets (Smalltalk par exemple),
- les noyaux de systèmes experts vides, fondés en général sur un formalisme unique de représentation (souvent les règles de production), par exemple OPS-5, Tango, S1, Cognitif, etc. ,
- les environnements logiciels complets fournissant à la fois un moteur d'inférence, une représentation multi-modes (avec notamment un formalisme d'objets structurés) et des logiciels d'interfaces très complets avec le cogniticien, l'expert et l'utilisateur final, par exemple ART, KEE, Savoir, TG2, etc.

1.4. Conclusion

La diversité des compétences et des besoins des intervenants du processus de production requiert qu'un soin particulier soit apporté à l'étude des moyens d'expression. Une bonne intégration des outils fournis par un environnement informatique est souhaitable notamment du fait que la structure de production peut amener une même personne à assumer plusieurs des tâches du cycle de vie.

Lors de la spécification du logiciel, l'écriture du scénario est souvent influencée par l'outil utilisé, par exemple un langage auteur. Nous avons critiqué les différents types de moyens d'expression proposés à l'auteur (dans une structure de production n°1) par les systèmes d'EAO: langage auteur (langage de programmation impératif "simplifié" et "habillé"), langage auteur "graphique" (organigrammes), langage "fonctionnel" (au sens composition des fonctions de différents éditeurs) "figé". A notre avis, il faut proposer aux auteurs des formes d'expression plus proche d'un *langage de spécification*, que d'un *langage de programmation* comme c'est le cas actuellement. En choisissant un *langage de spécification exécutable* , on favorise le prototypage.

L'évolutivité du logiciel pendant le développement n'est pas bien prise en compte par de nombreux systèmes auteurs qui ne facilitent pas la modification des logiciels développés. Il faut, par exemple, modifier directement le code du programme écrit en langage auteur, pour changer une analyse de réponse.

A notre avis, une bonne approche du problème, est celle choisie dans le système Diane-Editeur fonctionnel où un didacticiel est décomposé en termes d'objets EAO: scènes, textes, graphiques, animations, analyses de réponses, etc; accessibles et modifiables au travers d'éditeurs spécialisés. L'idée d'un développement incrémental est naturellement renforcée par cette "approche objet". Ceci est conforté par la possibilité d'utiliser des outils généraux de traitement de textes, de dessins (Mac Paint, PC Paint,

PC Brush, etc, sur micro-ordinateurs), d'images ([Mor86]), de sons ([Ben84]), ou encore des logiciels d'animations ou de production de dessins animés (VideōWorks).

Pour favoriser cette forme de conception par objets, il devient nécessaire de s'intéresser aux objets caractérisant un domaine d'enseignement particulier et de développer les éditeurs correspondants.

L'appropriation du logiciel par l'enseignant lors de l'exploitation est "une fonctionnalité peu courante sur le plan des systèmes d'EAO. Très souvent, les didacticiels sont "fermés" et n'autorisent l'enseignant utilisateur qu'à intervenir au niveau de la gestion et de l'édition de rapports d'évaluation ou de rapports administratifs" ([Mae84]). Il faut d'une part prévoir un grand degré de paramétrage du logiciel en vue de cette appropriation et d'autre part uniformiser les moyens d'expression fournis à l'enseignant, qu'il soit auteur ou utilisateur.

Ces réflexions nous ont guidé tout au long de notre expérimentation, au travers de la réalisation des divers logiciels présentés dans l'introduction. Le langage Prolog, *langage de spécifications exécutable* , sert de moyen d'expression unique pour l'informaticien et l'enseignant qu'il soit auteur ou utilisateur. La production s'appuie sur un modèle de cycle de vie avec *prototypage*, et *développement incrémental* par enrichissements successifs de différentes bases d'*objets EAO* et d'*objets du domaine* d'enseignement. Plus particulièrement, nous avons privilégié des domaines d'application ayant "une forte composante langage", les deux expérimentations effectués, ayant porté sur l'enseignement du Français langue étrangère et sur l'initiation à l'algorithmique.

Chapitre II

EAOLOG: UN PREMIER NIVEAU DE FONCTIONS E.A.O.

Le système Eaolog est un noyau de système de production construit sur un environnement Prolog (§ 2.1). Nous illustrons les moyens de spécification fournis à un auteur sur un exemple d'E.A.O. du Français Langue Etrangère (§ 2.2). Les différentes approches (grammaire, procédurale, relationnelle) du langage Prolog, permettent d'utiliser un moyen d'expression unique que ce soit pour la réalisation du système ou pour le développement d'un logiciel particulier (§ 2.3).

2.1. Fonctionnalités auteur

L'auteur dispose de modèles d'objets et d'actions associées. Il y accède par l'intermédiaire d'éditeurs appropriés, qui permettent d'instancier un objet et de le visualiser. Au travers de ces éditeurs, l'auteur manipule la description des objets dans un formalisme très proche de Prolog.

2.1.1. Moyens d'expression fourni à l'auteur

L'auteur compose des dialogues en termes d'objets d'interface EAO, et de grammaires décrivant les langages de réponses.

a) objets d'interface EAO

Les modèles d'objets fournis par système sont les suivants: points,

zones d'écran, messages, stimuli, scènes, menus. Les actions associées permettent leur présentation, leur réception et leur traitement.

Points, zones d'écrans et messages sont des objets au sens habituel.

La notion de stimulus englobe toute information proposée à l'apprenant au travers des media disponibles. Dans le contexte de notre expérimentation, nous nous limitons à l'écran du poste de travail. La description d'un stimulus comporte ainsi une liste de messages et les informations nécessaires à leur localisation sur l'écran et aux aspects d'affichage. Un stimulus peut être affiché ou effacé.

La notion de scène permet d'exprimer statiquement le passage d'un écran à un autre au cours du déroulement d'un dialogue. La description d'une scène comporte donc les informations nécessaires à la modification des stimuli apparaissant sur un écran: liste de stimuli à ajouter et/ou à effacer.

Un menu permet de solliciter un choix quant au déroulement du dialogue: dans notre expérimentation, nous nous limitons à une forme simple dans laquelle l'apprenant doit désigner une zone particulière de l'écran. Un menu comporte donc une scène, des zones de réception et les éléments permettant d'orienter la suite du dialogue en fonction du choix.

Les figures 2.1 et 2.2 montrent un exemple de description d'objets

Figure 2.1: modèle de description d'une scène

scène (Nom de la scène, Liste des noms de stimuli à effacer,
Liste des noms de stimuli à afficher)

stimulus(Nom du stimulus, Nom de la liste de messages, Nom de la
liste de points, Aspect d'affichage)

messages(Nom de la liste de messages, Liste des messages)

points-d'affichage(Nom de la liste de points, Liste des points).

Figure 2.2: exemple de scène

scène(scène1, [ste1], []). (1)

stimulus(ste1, lmsgs1, lpts1, souligné).

messages(lmsgs1, ["Les Vacances"]). (2)

points-d'affichage(lpts1, [(3,4)]). (3)

Dans l'expression 1, [] désigne la liste vide. Les expressions 2 et 3 décrivent la liste de messages et la liste de points d'affichage du stimulus **ste1**, comme étant le message "les Vacances" qui doit être affiché sur l'écran à partir du point de coordonnées 3 (3ème ligne), 4 (4ème colonne).

La bibliothèque fournit des actions de base applicables sur ces objets, pour l'envoi, la réception, et le traitement d'information.

Exemples

- envoi d'information:

  avancer-scène(scex): affichage de la scène de nom scex
  reculer-scène(scex): action inverse de la précédente
  afficher(stex): affichage du stimulus écrit de nom stex
  effacer(stex): effacement du stimulus écrit de nom stex

- réception d'information:

  réception(receptx, Msg-reçu): on réceptionne dans la zone de nom receptx, un message que l'on nomme Msg-reçu

- traitement d'information:

  msfc(Message-reçu, Message-msfc): on transforme le message-reçu en un message mis sous forme canonique message-msfc qui est la liste de mots du message-reçu

- temporisation et déroulement des dialogues:

  début_dialogue: on affiche à l'écran un certain format standard de début de dialogue propre à tout le didacticiel

  suite: on affiche à l'écran un message donné et on attend la réception d'un caractère au clavier pour passer à la suite.

b) grammaire d'un langage de réponse

La spécification d'une analyse de réponses consiste en la description, sous la forme d'une grammaire Prolog, d'un langage de réponses. La figure 2.3. est un exemple classique d'une telle grammaire.

figure 2.3. grammaire simplifiée d'une phrase en français

```
phrase --> sujet, verbe, complément.
sujet --> groupe_nominal.
complément --> groupe_nominal.
groupe_nominal --> article, adjectif, nom.
verbe --> ["mange"].
article  --> ["le"].
adjectif --> ["petit"].
nom --> ["chat"].
```

Dans cette grammaire, on retrouve:

-VN: le vocabulaire non terminal: {phrase, sujet, verbe, complément, groupe-nominal, article, adjectif, nom}

-VT: le vocabulaire terminal: {"mange", "le", "petit", "chat"}. Ces symboles terminaux sont notés entre [et] dans les règles de métamorphose.

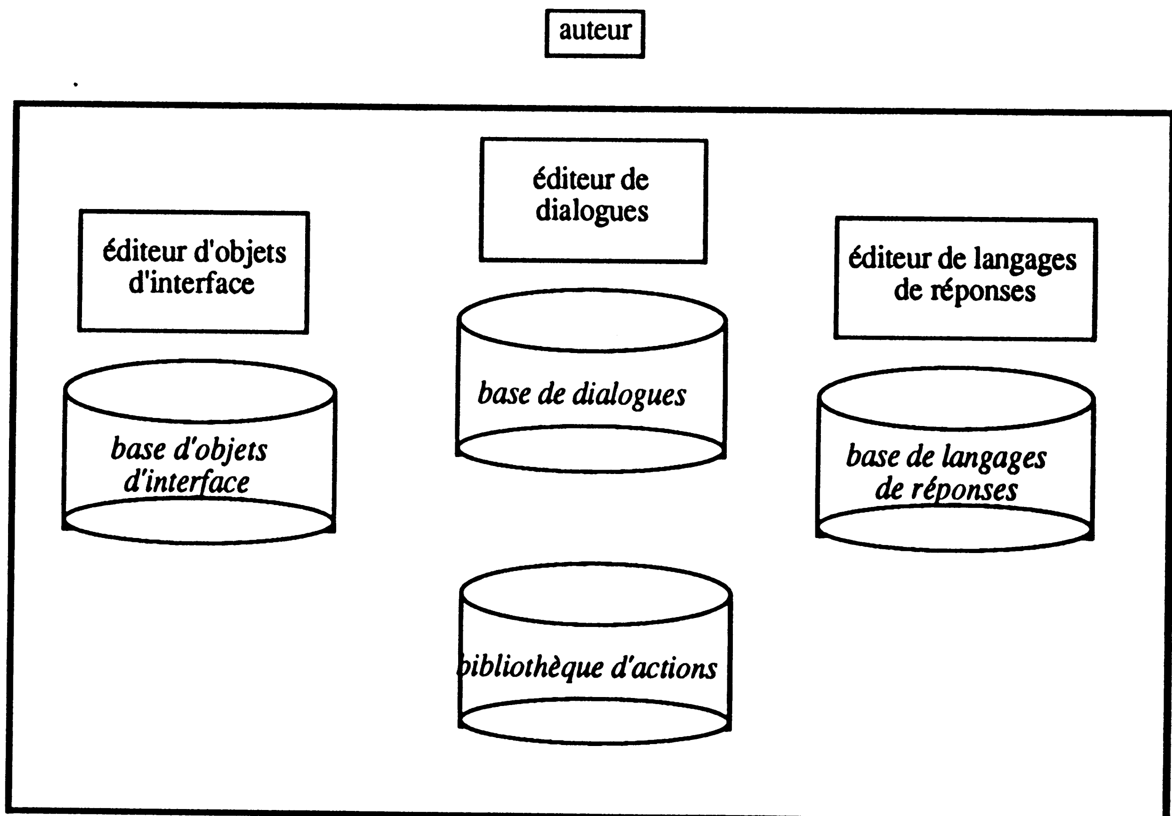
Nous illustrons au paragraphe 2.2. les étapes de spécification d'un langage de réponse autour d'un exemple concret.

2.1.2. Environnement d'édition

Le système doit permettre à l'auteur d'éditer les objets, c'est-à-dire les *créer*, les *supprimer*, les *modifier*, les *tester* et les *valider pédagogiquement*, et ceci indépendamment les uns des autres.

Les objets d'interface EAO sont décrits sous forme de faits. Les langages de réponse sont décrits sous forme de grammaire, c'est-à-dire de faits (vocabulaire) et de règles (syntaxe et sémantique). Les dialogues, qui composent les objets précédents, sont décrits sous la forme de règles. Nous avons ainsi défini trois éditeurs: pour les objets d'interface EAO, pour les langages de réponses et pour les dialogues (figure 2.4).

Figure 2.4: l'environnement d'édition du système Eaolog



Pour chaque type d'objet d'interface, l'éditeur fournit des fonctions classiques de: création, suppression, modification, consultation. De plus, il permet de visualiser les scènes et les menus.

Un langage de réponses est donné sous la forme d'une grammaire Prolog. On peut donc, à partir de la même grammaire, analyser des réponses ou générer tout ou partie du langage décrit. Cette deuxième fonction est nécessaire à l'auteur pour l'élaboration d'un langage des réponses, car elle lui permet d'examiner l'ensemble des phrases du langage décrit par la grammaire.

En ce qui concerne les dialogues, nous utilisons pour l'instant l'éditeur de clauses et l'interpréteur de l'environnement Prolog sur lequel nous travaillons.

2.2. Un exemple de dialogue

L'exemple que nous développons est tiré de l'un des didacticiels de Français Langue Etrangère du projet Mosaïque. Il s'agit du dialogue "les Vacances" [Luc 84], dont l'objectif est de faire travailler sur la syntaxe des phrases interrogatives. Nous l'utilisons pour illustrer en détail les étapes de spécification d'un langage de réponses. Le dialogue attendu est illustré par les écrans suivants: Un premier écran présente l'activité. Un deuxième écran spécifie les choix proposés à l'élève. Un dernier écran fournit un corrigé type.

On donne à l'apprenant un ensemble de verbes à l'infinitif (aller, partir, rester, habiter, voyager, revenir), un ensemble de mots interrogatifs (combien de temps, avec qui, comment, dans quel pays, quand, où) et implicitement deux sujets possibles (tu, vous). On donne de plus à l'apprenant une certaine réponse: en voiture, au Portugal, 3 semaines, avec des amis. Il doit trouver une forme correcte de phrase interrogative,

correspondant à la réponse proposée, en utilisant les mots à sa disposition.

Ecran1:

Les Yacances

M. Roche partira bientôt en vacances,

il a répondu aux questions d'un ami,

retrouvez ces questions en vous aidant des éléments donnés.

-->

Ecran2:

Les Yacances

combien de temps

avec qui comment

dans quel pays

quand où

aller partir rester habiter

voyager revenir

L'ami: ?

M. Roche: en voiture

-->

Les écrans suivants correspondent à l'écran2 avec un nouveau contexte de réponse: "au Portugal" au lieu de "en voiture" etc.

Le dialogue se termine par un écran corrigé donnant pour chaque contexte de réponse, une des phrases interrogatives possibles (voir écran final).

Ecran final

Les Vacances

L'ami: comment voyagerez vous ?
M. Roche: en voiture
L'ami: dans quel pays irez vous ?
M. Roche: au Portugal
L'ami: combien de temps resterez vous ?
M. Roche: 3 semaines
L'ami: avec qui partirez vous ?
M. Roche: avec des amis

-->

2.2.1. Description du langage de réponse

Nous développons ici la démarche proposée aux auteurs pour l'expression de l'analyse de réponse. Cette démarche correspond à la description par étapes d'un certain "langage de réponse" incluant les cas corrects et les cas erronés ("grammaire d'erreur"). Les différentes étapes correspondent aux descriptions successives

- au niveau du langage de réponse correct: de la syntaxe et du vocabulaire, des contraintes d'accords et des contraintes sémantiques ;

- au niveau de la grammaire d'erreur: des erreurs de syntaxe, des erreurs d'accords et de sémantique,
- dans les deux cas de la réaction à la réponse de l'apprenant sous la forme d'actions à entreprendre.

a) description de la syntaxe et du vocabulaire

Nous décrivons sous la forme de règles de grammaire, la syntaxe et le vocabulaire du langage de réponse correspondant à l'ensemble des phrases correctes attendues: (voir figures 2.5 et 2.6).

figure 2.5: Syntaxe

(1) phrase-interro --> mot-interro, verbe, sujet.
(2) phrase-interro --> mot-interro, sujet, verbe.
(3) phrase-interro --> mot-interro, part-interro, sujet, verbe.
(4) phrase-interro --> sujet, verbe, mot-interro.

La formule (1) exprime qu'une phrase interrogative est composée d'un mot interrogatif, suivi d'un verbe puis d'un sujet. Les formules (1), (2), (3), et (4) expriment les différentes formes d'une phrase interrogative.

figure 2.6: Vocabulaire

sujet --> [vous]. sujet --> [tu].
verbe --> [irez]. verbe --> [iras]. verbe--> [partirez].
...
mot-interro --> [avec, qui]. mot-interro-->[comment].
...
part-interro --> [est, ce, que].

b] description des contraintes contextuelles

Par rapport à la description syntaxique du langage, il faut ajouter des contraintes contextuelles; sur l'exemple de ce dialogue, il faut exprimer des

contraintes d'accords et des contraintes sémantiques pour refuser dans le langage des réponses correctes des phrases comme "quand partirez-tu", "quand habiterez vous" ou "comment voyagerez-vous" avec le contexte de réponse " au Portugal".

La description de ces contraintes se fait par l'introduction d'attributs dans les règles et par l'expression de vérifications d'accords et de vérifications sémantiques (voir figures 2.7 à 2.10).

figure 2.7: Contraintes d'accords (syntaxe)

(1) phrase-interro --> mot-interro, verbe(Nombre), sujet(Nombre).
(2) phrase-interro --> mot-interro, sujet(Nombre), verbe(Nombre).
(3) phrase-interro -->mot-interro, part-interro, sujet(Nombre), verbe(Nombre).
(4) phrase-interro --> sujet(Nombre), verbe(Nombre), mot-interro.

La formule (1) exprime qu'une phrase interrogative est composée d'un mot interrogatif, suivi d'un verbe et d'un sujet qui doivent être en accord en nombre.

figure 2.8: Contraintes d'accords (vocabulaire)

sujet(singulier) --> [tu]. sujet(pluriel) --> [vous].
verbe(singulier) --> [partiras]. verbe(pluriel) --> [partirez]. ...
mot-interro --> [quand]. ...

figure 2.9: Contraintes sémantiques (syntaxe)

(1) phrase-interro(Liensem) -->
mot-interro(Liensem), verbe(Liensem, Nombre), sujet(Nombre).
(2) phrase-interro(Liensem) -->
mot-interro(Liensem), sujet(Nombre), verbe(Liensem,Nombre).
(3) phrase-interro(Liensem) -->
mot-interro(Liensem), part-interro, sujet(Nombre),
verbe(Liensem,Nombre).
(4) phrase-interro(Liensem) -->

sujet(Nombre), verbe(Liensem,Nombre), mot-interro(Liensem).

Remarque: liensem2 correspond au contexte "en voiture".

Cette vérification sémantique est fondée sur la répartition des mots que l'apprenant doit utiliser en classes sémantiques. Chaque classe est caractérisée par un nom (liensem1, liensem2,...) qui apparait dans la description du vocabulaire.

La formule (1) exprime qu'une phrase interrogative est composée d'un mot interrogatif, suivi d'un verbe et d'un sujet qui doivent être en accord en nombre. On exprime de plus qu'il doit y avoir correspondance entre le lien sémantique du contexte (attribut hérité de phrase-interro) et les liens sémantiques du mot interrogatif et du verbe.

figure 2.10: Contraintes sémantiques (vocabulaire)

```
sujet(singulier) --> [tu].
sujet(pluriel) --> [vous].
verbe(liensem1,singulier) --> [partiras].
verbe(liensem1,pluriel) --> [partirez]. ...
mot-interro(liensem2) --> [comment]. ...
```

c] grammaire d'erreur et actions à entreprendre

Pour l'instant, nous n'avons décrit qu'un langage des réponses correctes: en mode analyse, on peut vérifier si une phrase donnée appartient ou non à l'ensemble des phrases correctes, en mode génération on peut connaître toutes les phrases de cet ensemble.

Les auteurs peuvent utiliser ce mode génération pour vérifier si la spécification du langage des réponses correctes est acceptable, en générant l'ensemble des réponses jugées correctes.

En mode analyse, la description effectuée permet uniquement de dire

si oui ou non une phrase donnée est correcte. Or en enseignement assisté par ordinateur, le cas où la phrase n'appartient pas au langage des réponses correctes est plus intéressant que l'autre et doit être traité en reconnaissant l'erreur faite.

C'est dans le cas d'une réponse incorrecte que l'on doit pouvoir aider et "évaluer" l'apprenant, en ayant compris le pourquoi de son erreur. Cette aide peut-être l'affichage d'un message d'erreur global (pour ne pas donner trop vite la solution) ou plus précis. On peut aussi donner un certain outil de correction de sa réponse, ou bien afficher la règle de grammaire mise en cause, ou encore offrir l'accès à une base d'informations, etc.

Il faut donc décrire la grammaire des règles correspondant aux configurations d'erreurs possibles et les actions à entreprendre associées. Celles-ci correspondent à l'aide que l'auteur souhaite donner à l'apprenant. Les actions d'évaluation de l'apprenant qui sont aussi associées aux configurations d'erreurs, peuvent être traitées de la même manière.

En donnant la possibilité d'associer des actions du répertoire aux règles de grammaires, nous passons d'un mode d'expression purement descriptif du langage de réponse à un autre où l'on met en relation les différents cas prévus par la grammaire et les actions à effectuer.

Pour simplifier la présentation de l'exemple, les actions à entreprendre (l'aide à donner à l'apprenant) sont des actions d'affichage de messages d'erreurs. Nous décrivons donc une "grammaire d'erreur", qui permet de décider quelles sont les actions à entreprendre. Nous passons ainsi de la description d'un langage des réponses correctes à la description du langage des réponses de l'apprenant.

figure 2.11: Extraits de la grammaire d'erreurs

```
phrase-interro(Liensem) -->
    mot-interro(Liensem), sujet(Nombre),
    afficher("il manque le verbe").
```

```
phrase-interro(Liensem) -->
    sujet(Nombre), mot-interro(Liensem),
    verbe(Liensem, Nombre), afficher("ordre incorrect").
...
phrase-interro(Liensem) -->
    sujet(Nombre1), verbe(Liensem, Nombre2),
    mot-interro(Liensem),verif-accords(Nombre1, Nombre2).
...
verif-accords(Nombre, Nombre).
verif-accords(Nombre1, Nombre2):- Nombre1 ≠ Nombre2,
afficher("erreurs d'accords").
```

Remarques:

- dans cette présentation, nous avons donné explicitement le texte des messages dans les actions d'affichage associées aux configurations d'erreurs. En réalité, on utilise des actions du type avancer-scène ou afficher-stimulus, paramétrées par le nom d'une scène ou d'un stimulus de la base d'objets d'interface.
- cette grammaire d'erreur peut être simplifiée, si l'on exprime les vérifications à effectuer, sur "l'arbre abstrait" de la phrase, préalablement construit. Nous reviendrons sur ce point au paragraphe 2.3.2.

2.2.2. Description des objets d'interface et du dialogue

Suivant le même schéma de présentation, on introduit l'expression de la composition des actions de la bibliothèque. On exprime par exemple, qu'un dialogue est composé d'un début de dialogue, d'un écran1, d'un écran2, d'un écran3 et d'une fin de dialogue; et que l'écran1 est composé d'un afficher(stimulus1), d'un afficher(stimulus2), etc.

```
dialogue :- début-dialogue, écran1, écran2, écran3, fin-dialogue.
    écran1 :-
        afficher(stimulus1), afficher(stimulus2),
        afficher(stimulus3), suite.
    écran2 :- effacer(stimulus3), afficher(stimulus4), suite.
    écran3 :- effacer(stimulus4), afficher(stimulus5), suite.
```

Pour l'exemple développé ici, le déroulement du dialogue est ainsi exprimé en Prolog par les règles de la figure 2.12:

figure 2.12: Dialogue "les Vacances"

```
dialogue :-
    début-dialogue, ava-scène(scène1), ava-scène(scène2),
    ava-scène(scène3), corps-dialogue, ava-scène(scène4), fin-dialogue.
corps-dialogue :-
    listes-d-exercices([contexte1, contexte2, contexte3,
        contexte4, contexte5, contexte6])
    listes-d-exercices( [] ).
    listes-d-exercices( [Contexte | SuiteContexte] ) :-
        exercice(Contexte), liste-d-exercices(SuiteContexte).
exercice(Contexte) :-
    description contexte(Contexte, Ste, Cont),
    afficher(Ste),
    réception(recept, Reponse),
    msfc(Reponse, Reponsemsfc),
    analyse(Cont, Reponsemsfc, Resultat),
    si Resultat = ok alors effacer(ste) sinon exercice(Contexte) fsi.
```

Remarque: nous avons souligné les éléments ne faisant pas partie du répertoire d'actions primitives. Ces éléments sont utilisés par les enseignants auteurs pour structurer leur dialogue.

En utilisant les notations définies précédemment nous donnons un extrait des objets d'interface manipulés dans ce dialogue.

figure 2.13: objets d'interface du dialogue "les Vacances"

<u>scènes</u>

scène(scène1, [], [ste1,ste2])
scène(scène2, [ste2], [ste3,ste4,ste5,ste6,ste7])
...

<u>stimuli</u>

stimulus(ste1, lmsgs1, lpts1, asp(0,12,4))
stimulus(ste2, lmsgs2, lpts2, asp(0,12,0))
...

<u>messages</u>

messages(lmsgs1, ["Les Vacances"])
messages(lmsgs2, ["M. ROCHE partira bientôt en vacances", "il a répondu aux questions d'un ami", "Retrouvez ces questions en vous aidant des éléments donnés"])
...
messages(coe1, ["cette phrase est correcte"])
messages(coe2, ["oubli du mot interrogatif"]).
messages(coe3, ["oubli du sujet et du mot-interrogatif"]).
messages(coe4, ["erreur d'accord entre le sujet et le verbe"])
messages(coe5, ["erreur en ce qui concerne le verbe choisi"])
...

<u>points d'affichage</u>

points-d'affichage(ste1, [(3,4)])
points-d'affichage(ste2, [(7,6), (8,7), (9,8)])
...

<u>zones de reception</u>

zones(recept,[(16,38,16,70)]).
...

2.3. Prolog et la production de logiciels pour l'enseignement.

Nous présentons ici les différentes approches du langage Prolog qui nous ont servi pour réaliser le système Eaolog. La bibliothèque d'actions de base a été réalisée selon une approche procédurale (§2.3.1). L'approche relationnelle est à la base de la réalisation des éditeurs (§2.3.2).

Nous avons déjà vu les apports de l'approche grammaire pour la spécification des langages de réponses. Nous prolongeons ici cet aspect (§2.3.3) en montrant que l'on peut spécifier les objets propres au domaine d'enseignement selon l'approche grammaire. Ceci est à la base du système Tangram présenté dans la suite de la thèse.

Le langage Prolog a été développé par A. Colmerauer et P. Roussel ([Col75], [Rou75]), en collaboration avec R. Kowalski([Kow79]), à partir des travaux des logiciens, et particulièrement ceux de A. Robinson ([Rob65]). Ce langage est maintenant utilisé, notamment selon les approches citées ci-dessus, dans divers domaines de l'informatique: génie logiciel, écriture de compilateurs, traitement des langues naturelles, calcul symbolique, bases de données, systèmes experts, construction de plans en robotique ([War 80], [Sim 81], [Dah 77], [BBL79], [Din 79], [DLL81]).

L'interpréteur Prolog se comporte comme un démonstrateur de théorèmes reposant sur le principe de résolution de Robinson avec des restrictions correspondant à l'utilisation de clauses de Horn. Les clauses de Horn sont des clauses contenant au plus un littéral positif. Elles sont donc de la forme:

$$B_n \text{ ou } \neg A_1 \text{ ou } \neg A_2 \text{ ou } \ldots \text{ ou } \neg A_p \quad (\text{avec } 0 \leq n \leq 1 \text{ et } p \geq 0)$$

ce qui peut être écrit

$$B_n <= A_1 \text{ et } A_2 \text{ et } \ldots \text{ et } A_p$$

après avoir utilisé l'équivalence: $(A => B) <=> (\neg A \text{ ou } B)$

On distingue 4 formes pour une clause de Horn:

règle	B <= A_1 et A_2 et ... et A_p	$n = 1$	$p > 0$
axiome	B <=	$n = 1$	$p = 0$
but	<= A1 et A_2 et ... et A_p	$n = 0$	$p > 0$
clause vide	Nil	$n = 0$	$p = 0$

Ces formes sont interprétées de la manière suivante:

règle	B est impliqué par A_1 et A_2 et ... et A_p
axiome	B est vrai
but	prouver A_1 et A_2 et ... et A_p (démonstration à effectuer)
clause vide	arrêt de la démonstration

La syntaxe du langage Prolog utilisée ici, et pour tous les exemples qui suivront, est celle de la version d'Edimbourg [ClM 81].

règle	B :- A_1 , A_2 , ... , A_p.	$n = 1$	$p > 0$
axiome	B.	$n = 1$	$p = 0$
but	A_1 , A_2 , ... , A_p.	$n = 0$	$p > 0$

Pour une présentation complète du langage Prolog par la logique nous renvoyons à [Del 85] qui, partant de la logique mathématique, présente les mécanismes de la programmation Prolog.

2.3.1. Réalisation de la bibliothèque de primitives E.A.O.: approche procédurale

La bibliothèque d'actions est organisée en plusieurs niveaux
- le premier correspond à une réécriture (changement de noms) des prédicats prédéfinis de l'environnement Prolog support. Ceci doit faciliter la portabilité du système.
- un deuxième niveau correspond à des fonctions et actions non spécifiques à l'EAO, par exemple des primitives de manipulation de listes, d'ensembles ou d'arbres.
- enfin un troisième niveau correspond aux fonctions de présentation, de réception et de traitement d'objets d'interface EAO, scènes, menus, stimuli, messages, zones d'écran, points...

La tâche de création et d'extension de cette bibliothèque est à la charge d'un informaticien. Comme nous le montrons dans les exemples qui suivent, il s'appuie pour cela sur une approche procédurale du langage Prolog.

Il est possible d'associer aux 4 formes d'une clause de Horn, une interprétation procédurale (voir par exemple [KoE 76]):
a) la règle A :- B_1, B_2, ... B_n. , est interprétée comme une déclaration de procédure. La conclusion A est le nom de la procédure et l'antécédent B_1, B_2, ... B_n en est le corps. Celui-ci consiste en une suite d'appels B_i. Les paramètres éventuels sont les termes qui sont les arguments de A et des B_i.
b) l'axiome A. est considéré comme une procédure dont le corps est vide.
c) le but B_1, B_2, ... B_n. est interprété comme l'appel des procédures correspondantes.
d) la clause vide *Nil* est interprétée comme un ordre d'arrêt.

a) exemples de primitives d'envoi d'information

La primitive **avancer_scène**, paramétrée par un nom de scène, s'écrit comme suit en Prolog:

```
        avancer-scène(N_sce) :-
(1)         scène(N_sce, L_sti1, L_sti2),
(2)         aff-lsti(L_sti1, mode_eff),
(3)         aff-lsti(L_sti2, mode_aff).
```

(1): recherche dans la base de scènes, de la scène de nom **N_sce**; on récupère les listes de stimulis à effacer **L_sti1**, et à afficher **L_sti2**.
(2): on utilise la primitive **aff-lsti** en mode effacement, pour effacer la liste de stimulis **L_sti1**.
(3): on utilise la primitive **aff-lsti** en mode affichage, pour afficher la liste de stimulis **L_sti2**.

La primitive **reculer_scène** s'écrit:

```
reculer-scène(N_sce) :-
    scène(N_sce, L_sti1, L_sti2),
    aff-lsti(L_sti1, mode_aff),
    aff-lsti(L_sti2, mode_eff).
```

En ce qui concerne la primitive **aff-lsti** son écriture repose sur une analyse récurrente sur la structure de liste:

```
aff-lsti([], M_ode) :- !.                          (base de la récurrence)
aff-lsti([T_ête | Q_ueue], M_ode) :-               (étape récurrente)
    aff-sti(T_ête, M_ode),
    aff_lsti(Q_ueue, M_ode).
```

b) exemple de primitive de traitement d'information

Nous montrons maintenant comment décrire en Prolog une primitive d'analyse de réponse que nous nommons **comparer-phrase**. Cette

primitive a pour données une phrase de l'apprenant et une phrase attendue par l'auteur. Elle doit détecter les différences entre la phrase donnée et la phrase attendue. Son format est le suivant:

comparer-phrase(Phd, Pha, Coe, Liste-mots-oubliés, Liste-mots-non-reconnus)

données

- Phd est la liste des mots de la phrase donnée,
- Pha est la liste des mots de la phrase attendue.

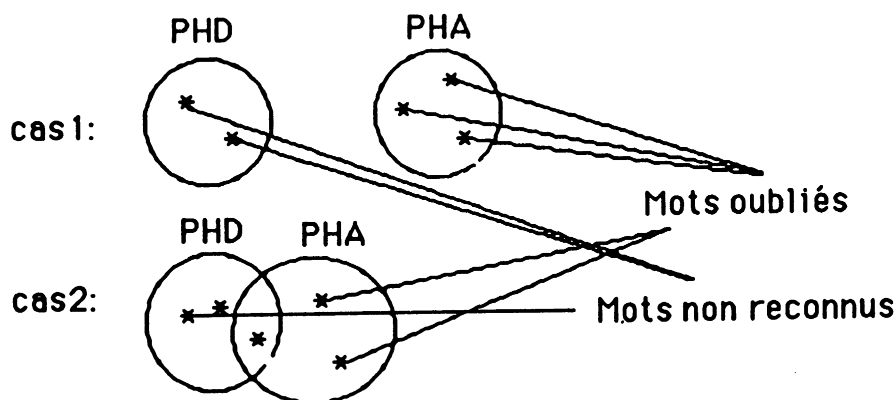
Dans chacune des listes de mots, un même mot n'apparait qu'une seule fois.

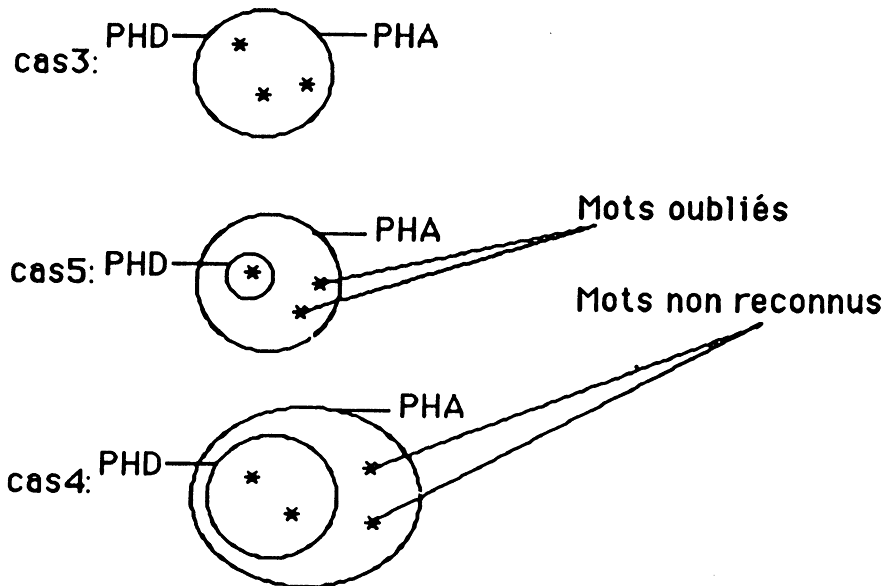
résultats

- Coe qui vaut: coe1 si la phrase est correcte, coe2 si l'ordre est incorrect, coe3 s'il y a des mots oubliés, coe4 s'il y a des mots non reconnus, coe5 s'il y a des mots non reconnus et des mots oubliés
- Liste-mots-oubliés est la liste des mots oubliés.
- Liste-mots-non-reconnus est la liste des mots non reconnus.

Les deux ensembles mots donnés et mots attendus étant donnés sous la forme de la liste de leurs éléments, nous faisons une analyse par cas sur la position relative de ces deux ensembles.

Nous avons les différents cas suivants:





cas1: disjoint(PHD,PHA): la liste de "mots oubliés" est la liste PHA, la liste de "mots non reconnus" est la liste PHD.

cas3: ce cas se décompose en 2 sous-cas:

sous-cas3.1: égalité(PHD,PHA): dans ce cas, la phrase est correcte.

sous-cas3.2: égalité-ens(PHD,PHA) et égalité(PHD,PHA): ici l'ordre est incorrect"

cas4: sous-ens(PHD,PHA) et égalité-ens(PHD,PHA):

la liste de "mots oubliés" est la liste R obtenue par diff-ens(PHA,PHD,R).

cas5: sous-ens(PHA,PHD) et égalité-ens(PHD,PHA):

la liste de "mots non reconnus" est la liste R obtenue par diff-ens(PHD,PHA,R).

cas2: la liste de "mots oubliés" est la liste R1 obtenue par intersection(PHA,PHD,X), diff-ens(PHA,X,R1), la liste de "mots non reconnus" est la liste R2 obtenue par diff-ens(PHD,X,R2).

On obtient ainsi le programme Prolog suivant:

```
comparer-phrase(Phd, Pha, coe1, [], []) :- égalité-liste(Phd, Pha), !.
comparer-phrase(Phd, Pha, coe2, [], []) :- égalité-ens(Phd, Pha), !.
comparer-phrase(Phd, Pha, coe3, X, []) :-
    sous-ens(Phd, Pha), diff-ens(Pha, Phd, X), !.
comparer-phrase(Phd, Pha, coe4, [], X) :-
    sous-ens(Pha, Phd), diff-ens(Phd, Pha, X), !.
comparer-phrase(Phd, Pha, coe5, Pha, Phd) :- disjoint(Phd, Pha), !.
comparer-phrase(Phd, Pha, coe6, Y, Z) :-
    intersection(Phd, Pha, X), diff-ens(Pha, X, Y), diff-ens(Phd, X, Z).
```

2.3.2. Réalisation des éditeurs: approche relationnelle

La réalisation des éditeurs consiste à implémenter des fonctions d'interrogation, d'ajout, de modification et de suppression de faits ou de règles. Elle tient compte de la structure des objets à éditer. Nous nous sommes appuyés pour cela sur une approche relationnelle de Prolog, que nous présentons ici.

Cette approche sert souvent d'exemple introductif à Prolog, en montrant que ce langage permet de déclarer des faits, de définir des règles et de poser des questions. Par exemple dans [CKV 83] Prolog est introduit comme un langage d'expression d'un univers correspondant à un ensemble d'objets et d'un ensemble de relations décrivant les propriétés des objets et leurs interactions. Pour l'exemple choisi dans cet ouvrage, les objets sont les mots proposés à la carte d'un restaurant. Un ensemble de relations donne la classification de ces mots en hors-d'oeuvre, viande, poisson et dessert.

Une base de données relationnelle est constituée d'un ensemble de relations de la forme: $R(A_1, A_2,...,A_n)$, où R est le nom de la relation et

$A_1, A_2,\ldots,A_n$ les noms des attributs prenant leurs valeurs dans leurs domaines respectifs. L'expression, en Prolog de la même base de données est décrite par un ensemble de clauses de la forme: $P(A_1, A_2,\ldots,A_n)$, où P est le nom d'un prédicat et $A_1, A_2,\ldots,A_n$ les noms des variables représentant les attributs $A_1, A_2,\ldots,A_n$.

Nous retrouvons donc les 3 formes d'une clause de Horn, avec une nouvelle interprétation:
a) l'axiome *A.* permet de donner une relation en extension par un ensemble de faits.
b) la règle A :- $B_1, B_2,\ldots, B_n$. définit une nouvelle relation (virtuelle) à partir d'autres relations.
c) le but$B_1, B_2,\ldots, B_n$. représente une question.

Prolog permet ainsi dans un même formalisme, de représenter et d'interroger les informations que l'on a sur un domaine.

Au niveau de l'expression des règles et des questions, Prolog est très proche des langages prédicatifs d'accés à des bases de données relationnelles. En effet, ces langages prédicatifs sont fondés sur la logique mathématique tout comme Prolog l'est sur la théorie du premier ordre, il est donc facile d'exprimer en Prolog des requêtes formulées en langage prédicatif à variable domaine.

En Prolog, les primitives de gestion sont exprimées en utilisant des prédicats prédéfinis de gestion de clauses (ajout, suppression, modification de clauses).

Prolog peut donc être vu comme le noyau d'un système de gestion de base de données (SGBD) relationnel. Il permet à la fois de représenter les relations de la base, et de les manipuler. D'autre part, alors que le langage de manipulation de données d'un SGBD classique est encapsulé dans un

langage hôte (Cobol par exemple), avec Prolog c'est lui-même qui sert de langage hôte, ce qui fait disparaitre l'hétérogénéité précédente. Au niveau de la représentation des relations, une base de données est représentée par un ensemble de clauses Prolog. Au niveau de la manipulation des données, Prolog est un langage relationnellement complet: toute expression de l'algèbre relationnelle peut être exprimée en Prolog.

Par contre "les implantations actuelles de Prolog, à deux exceptions près ([Don 83], [Nai 83]), ne permettent pas de manipuler directement des clauses en mémoire secondaire. L'ensemble du programme Prolog (et en particulier les faits) doit se trouver dans la mémoire active. Ceci est évidemment une grave lacune pour l'utilisation industrielle de Prolog comme SGBD." ([Cnd 86]).

2.3.3. Apports de l'approche grammaire

C'est en 1970, qu' Alain Colmerauer conçoit un langage de programmation non-déterministe: les systèmes-q ([Col70]). Ce formalisme permettant de décrire des grammaires complexes, auquel était associé un interpréteur pour analyser ou synthétiser des structures conformes à ces grammaires. La base du formalisme était des règles de réécriture.

Par la suite, A. Colmerauer participe à l'élaboration du langage Prolog, et développe les grammaires de métamorphose: il s'agit d'une axiomatisation en logique du premier ordre de la façon de traiter l'associativité de la concaténation, lorsque l'on travaille sur des chaines, afin de disposer en Prolog, des facilités que donnent les systèmes-q, et d'obtenir ainsi un outil pour le traitement syntaxique et sémantique des langages.

Une définition de ces grammaires, sous forme de système de réécriture est donnée dans ([Col75]). Les grammaires de métamorphose permettent d'écrire des grammaires hors contexte.

Dans les systèmes Prolog, permettant d'utiliser cette notation pour les règles de grammaire, ces règles sont transformées sous forme de clauses Prolog ordinaires. Cette transformation se fait en créant une clause pour chaque règle et un prédicat pour chaque non terminal, tandis que les terminaux sont incorporés aux paramètres ajoutés aux prédicats créés.

De façon pratique, pour chaque non terminal *nt(X,Y)* comportant les arguments *X* et *Y*, lorsqu'il est transformé en prédicat Prolog, on lui ajoute deux arguments *Ce* et *Cr*, qui sont la chaine d'entrée *Ce* avant analyse et ce qui en reste, *Cr* après analyse.

A titre d'exemple, nous donnons les clauses Prolog correspondant à une grammaire simplifiée du français (cf figure 2.3, §2.1.1 b):

```
phrase(Ce, Cr) :- sujet(Ce, C1), verbe(C1, C2), complément(C2, Cr).
sujet(Ce, Cr) :- groupe_nominal(Ce, Cr).
complément(Ce, Cr) :- groupe_nominal(Ce, Cr).
groupe_nominal(Ce, Cr) :- article(Ce, C1), adjectif(C1, C2), nom(C2, Cr).
verbe(["mange" | Cr], Cr).
article(["le" | Cr], Cr).
adjectif(["petit" | Cr], Cr).
nom(["chat" | Cr], Cr).
```

La transformation qui permet de passer d'une grammaire à un programme Prolog peut facilement s'écrire en Prolog même. C'est ce qui est réalisé par l'éditeur de langages de réponses.

Les travaux sur la formalisation de la syntaxe et de la sémantique des langues naturelles ([Cho59]) et des langages de programmation ([Knu68b]) depuis une vingtaine d'années ont mis en évidence une forme de *programmation par grammaire* qu'il est possible d'appliquer dans de nombreux domaines, en particulier quand les traitements à effectuer sont fortement dirigés par leurs données.

Prolog est adapté à ce style de programmation puisqu'il permet l'expression de règles d'analyse-génération et de construction de structures, ainsi que de règles de transformation, interprétation et visualisation de structures. Nous montrons sur un exemple simple quelques principes de base de la programmation par grammaire et leur expression en Prolog.

a) analyse et génération

La grammaire Prolog suivante correspond à la description de la syntaxe et du vocabulaire d'un langage de phrases affirmatives.

syntaxe du langage

```
phrase --> syntagme-nominal, syntagme-verbal.
syntagme-nominal --> article, adjectif, nom.
syntagme-verbal --> verbe, syntagme-nominal.
```

vocabulaire

```
article --> ["le"].          article --> ["la"].
adjectif --> ["petit"].      adjectif --> ["petite"].
nom --> ["chat"].            nom --> ["souris"].
```

Ce programme Prolog est utilisable de deux manières: en mode analyse il permet de reconnaître si une phrase donnée appartient ou non au langage décrit, en mode génération il permet d'obtenir l'ensemble des phrases du langage.

Pour prendre en compte des contraintes contextuelles, par exemple ici les vérifications d'accords en genre et en nombre, on ajoute des attributs supplémentaires aux règles du programme précédent.

```
phrase --> syntagme-nominal(n), syntagme-verbal(n).
syntagme-nominal(n) --> article(g,n), adjectif(g,n), nom(g,n).
syntagme-verbal(n1) --> verbe(n1), syntagme-nominal(n2).
article(masc,sing) --> ["le"].
article(fem,sing) --> ["la"].                         etc
```

b) construction de l'arbre abstrait

Partant de la même grammaire on peut écrire un programme qui construit l'arbre abstrait des phrases du langage.

L'arbre abstrait de la phrase "le petit chat mange la petite souris"est:

```
ph(sn(art(le),adj(petit),nm(chat)),sv(vb(mange),sn(art(la),adj(petite),nm(souris))))
```

Le programme qui construit cet arbre abstrait est le suivant:

```
phrase(ph(a1,a2)) --> syntagme-nominal(a1), syntagme-verbal(a2).
syntagme-nominal(sn(a1,a2,a3)) --> article(a1), adjectif(a2), nom(a3).
syntagme-verbal(sv(a1,a2)) --> verbe(a1), syntagme-nominal(a2).
article(art(le)) --> ["le"].
article(art(la)) --> ["la"].                          etc.
```

c) transformation, interprétation et visualisation

On peut ensuite écrire des programmes de visualisation, d'interprétation ou de transformation de l'arbre abstrait construit.

Par exemple, supposons que l'on veuille permuter le syntagme nominal complément et le syntagme nominal sujet. Ainsi la phrase "le petit chat voit la petite souris" devient "la petite souris voit le petit chat". La règle Prolog correspondant à cette transformation est:

```
transformation(ph(a1,sv(a2,a3)),ph(a3,sv(a2,a1))).
```

Si l'on veut visualiser la structure de la phrase en utilisant l'indentation, la phrase

le petit chat mange la petite souris

peut être visualisée sous la forme suivante:

phrase
    syntagme nominal
        article: *le*
        adjectif: *petit*
        nom: *chat*
    syntagme verbal
        verbe: *mange*
        syntagme nominal
            article: *la*
            adjectif: *petite*
            nom: *souris*

Le programme Prolog effectuant cette visualisation, à partir de l'arbre abstrait de la phrase, est le suivant:

```
visualisation(ph(a1,a2),c1) :-
    pos-curseur-col(c1), ecrire("phrase"),
    c2 est c1 + 4, ligne,
    visualisation(a1,c2), visualisation(a2,c2).
visualisation(sn(a1,a2,a3),c1) :-
    pos-curseur-col(c1), ecrire("syntagme nominal"),
    c2 est c1 + 4, ligne,
    visualisation(a1,c2), visualisation(a2,c2), visualisation(a2,c3).
visualisation(sv(a1,a2),c1) :-
    pos-curseur-col(c1), ecrire("syntagme verbal"),
    c2 est c1 + 4, ligne,
    visualisation(a1,c2), visualisation(a2,c2).
visualisation(art(a),c1) :-
    pos-curseur-col(c1), ecrire("article:"), ecrire(a), ligne.
visualisation(nom(a),c1) :-
    pos-curseur-col(c1), ecrire("nom:"), ecrire(a), ligne.
visualisation(verbe(a),c1) :-
    pos-curseur-col(c1), ecrire("verbe:"), ecrire(a), ligne.
```

L'action **ligne** provoque un saut de ligne, l'action **pos-curseur-col(c-ol)** provoque un positionnement à la colonne **c-ol.**

2.4. Conclusion

La conception du système Eaolog s'est appuyée, comme nous l'avons dit sur une expérimentation en Français Langue Etrangère.

Les enseignants, n'ayant eu qu'une sensibilisation à l'informatique, ne savaient pas et ne voulaient pas écrire des programmes. Par contre, étant linguistes de formation, ils étaient habitués au formalisme des grammaires, en particulier sous la forme de règles de réécriture.

Nous leur avons présenté Prolog selon une approche grammaire. Dans un premier temps, nous nous sommes centrés sur la spécification des analyses de réponses, sous forme de langages de réponses. Nous avons pu ensuite leur montrer que cette description était aussi un programme exécutable et que l'on pouvait l'utiliser, soit en analyse, pour reconnaître qu'une phrase donnée appartient ou non au langage de réponse, soit en génération, pour produire l'ensemble des phrases acceptées par le langage. Finalement, il a été facile pour les auteurs de décrire les autres composants d'un dialogue dans ce même formalisme.

Cette expérience a ouvert des perspectives, en brisant une partie de l'auto-censure habituelle des auteurs. Leur démarche est en effet souvent "timide", ne cernant pas la puissance des outils mis à leur disposition. Le fait que les auteurs s'exprimaient sur un terrain connu, a donné plus de champ à leur imagination pour la création des dialogues.

Ainsi le moyen d'expression proposé semble facile d'accès aux enseignants. Il favorise une attitude méthodologique orientée sur la spécification des objets et non sur la visualisation a priori de l'exécution du dialogue que l'on conçoit. Le travail de l'enseignant est facilité aussi bien par les fonctionnalités interactives du système que par les modèles d'objets qu'il comporte. Il peut d'autre part utiliser des éditeurs d'objets non

spécifiques à l'EAO: éditeurs de dessins, d'animation,... Dans Eaolog, nous pouvons par exemple faire appel à des dessins édités sous MacPaint, pour élaborer un dialogue.

Du point de vue des informaticiens, le même moyen d'expression fournit toute la puissance nécessaire pour réaliser les éditeurs du système et les bibliothèques d'actions.

La démarche que nous avons employée peut être appliquée pour spécifier la structure d'un didacticiel (selon la terminologie Diane énoncée au chapitre 1). De plus, elle favorise la spécification des objets propres au domaine d'enseignement. Par exemple dans le contexte du Français Langue Etrangère, l'objet langage de réponse fait partie d'une modélisation des connaissances sur la langue (syntaxe, sémantique, règles de visualisation des phrases).

Pour évaluer cette approche, dans d'autres domaines en E.A.O., nous avons choisi l'enseignement de l'algorithmique et de la programmation. Ce domaine d'expérimentation est traité dans la deuxième partie de cette thèse.

Chapitre III

ANALYSE PEDAGOGIQUE DU LOGICIEL IDALGO

La deuxième partie de cette thèse présente le projet Tangram qui traite d'une spécialisation et d'une expérimentation du système de production Eaolog, dans le contexte de l'enseignement de l'algorithmique et de la programmation.

Dans ce chapitre, nous précisons le contexte d'expérimentation, en rappelant des exemples de logiciels pour l'enseignement de la programmation, et en présentant le contexte pédagogique d'enseignement (§ 3.1).

Nous décrivons ensuite le cahier des charges pédagogiques du logiciel Idalgo (§ 3.2), qui sert de base à notre expérimentation de production de logiciels pour l'enseignement de la programmation. Le cadre est celui d'une initiation à l'algorithmique qui introduit des concepts autour de notations précises et des méthodes autour de schémas d'algorithmes. Nous présentons un certain nombre d'activités pédagogiques susceptibles d'être pratiquées par l'apprenant sous contrôle du logiciel visé. Ce sont des activités d'écriture et de lecture d'algorithmes: elles concernent la formalisation algorithmique, le processus d'analyse (en amont) et les processus d'exécution et de codage (en aval). Nous terminons, en montrant la variété des formes de présentation des informations présentées dans le cahier des charges.

3.1. Contexte de l'expérimentation

3.1.1. Exemples de logiciels pour l'enseignement de la programmation

En EAO de la programmation, on retrouve les différents types de logiciels d'enseignement présentés au chapitre I: tutoriel, logiciel informatif, logiciel de simulation, tuteur intelligent et environnement d'apprentissage.

a) tutoriels , logiciels informatifs et logiciels de simulation

La majorité des logiciels commercialisés sont de type tutoriel et portent sur la syntaxe des langages (Basic, Pascal, Ada, C ...).

Le projet **Satire** (Système d'**A**dministration et de **T**ransmissions d'**I**nformations **R**elevant de l'**E**nseignement) présenté par M. Quéré dans ([Que80]), décrit un logiciel qui permet la consultation en modes tutoriel, adaptation, documentaire et récapitulatif, de concepts concernant une méthode d'analyse (méthode déductive [PaM75], [Pai79]), un langage algorithmique et différents langages de programmation: Pascal, Lse et Basic. L'environnement Satire communique avec l'environnement du système hôte comprenant des interpréteurs et compilateurs des différents langages présentés: compilateur Snoopy (pour le langage algorithmique), compilateurs Pascal et Lse et interpréteur Basic.

Le systeme **Satec** [Bar79] (Système d'Apprentissage des Techniques de Compilation) propose un outil d'illustration d'un cours de compilation. Il permet à l'apprenant entre autres de suivre les différents traitements effectués sur un programme durant sa compilation et de présenter une visualisation dynamique de l'exécution.

Le projet **Balsa** (**B**rown University **AL**gorithm **S**imulator and **A**nimator), présenté par M.H. Brown et R. Sedgewick dans [BrS83] et [BrS84], consiste à développer un laboratoire d'enseignement de la

programmation basé sur l'idée d'animation de programmes. L'utilisateur dispose de vues multiples des programmes et des structures de données associées. Les exemples de programmes "animés" donnés dans les différents articles concernent les arbres binaires (visualisation de la construction d'un arbre binaire, recherche dans un arbre binaire), l'analyse syntaxique d'expressions (visualisation de la correspondance entre les appels de procedures récursives et l'arbre d'analyse de l'expression), recherches dans un graphe, simplex, etc.

b) tuteurs intelligents, détecteurs d'anomalies et environnements de programmation pour débutants.

Dans [BoS87], B. du Boulay et C. Sothcott présentent un ensemble de travaux d'E.I.A.O. dans le domaine de la programmation, en séparant les *tuteurs intelligents*, les *détecteurs d'anomalies* et les *environnements de programmation pour débutants*.

Les *tuteurs intelligents* prennent des décisions sur comment et dans quel ordre les concepts de programmation doivent être introduits et tentent de guider la progression de l'apprenant dans la maîtrise des nouvelles notions (exemples: Phenarete [Wer85], Malt [KoB75], Spade [Mil78], Lisp Tutor, Capra et Saida).

Lisp Tutor ([AnR85]), par exemple, est un tuteur pour l'enseignement de la programmation en Lisp. Il dispose d'une base de connaissances correctes et d'une base de connaissances sur les erreurs, exprimées sous la forme de règles de production (approximativement 325 règles concernant la construction et l'écriture de programmes Lisp et 475 versions erronées de ces règles). L'apprenant dispose pour l'écriture de ses programmes Lisp d'un éditeur structuré. LispTutor (écrit en Franz_Lisp) tourne sur Vax et est commercialisé.

Saida ([Gra88]) (**Système d'Aide à l'Implantation de Données Abstraites**) est un prototype d'environnement de construction de programmes Ada destiné à l'enseignement et doté d'un système expert d'aide au choix d'implantations physiques pour les structures de données abstraites proposées dans la bibliothèque du système.

Capra ([FDV88]) est un projet de tuteur intelligent pour l'enseignement de la programmation (à un niveau initiation) s'appuyant sur l'idée de *schémas* pour raisonner sur la construction d'algorithmes. Cette idée de raisonner à partir de *schémas* ou *plans* est inspirée des travaux en méthodologie de la programmation ([LPS83], [ScP88]) et en intelligence artificielle ([Sol86]). Nous reviendrons sur ce sujet au paragraphe 3.1.2.

Les *détecteurs d'anomalies* ont eux, l'objectif plus restreint de localiser et de commenter les erreurs dans des programmes écrits par l'apprenant (exemples: Laura [Adl80], Mycroft [Gol75], Shapiro's system [Sha82] et Proust).

Proust ([JoS85]) est un détecteur d'anomalies qui cherche les fautes non syntaxiques dans les programmes Pascal écrits par des programmeurs débutants. Ce système analyse les erreurs de l'apprenant en comparant sa solution à des déviations de schémas de solutions. Proust (écrit en Lisp) tourne sur Vax, et est commercialisé.

Les *environnements de programmation pour débutants*, comprennent des outils tels que des éditeurs syntaxiques, des afficheurs, des metteurs au point, des traducteurs conçus pour des débutants (exemples: Bip [BBA76], Macintosh Pascal [Vos84] et Bridge).

Bridge ([Boc86]) est un système permettant la réalisation de programmes Pascal en trois étapes. Dans la première étape, l'apprenant doit construire une solution d'un problème posé en langue naturelle, sous la forme d'une séquence de buts pris dans un menu de phrases. Dans la

deuxième étape, l'apprenant doit associer à chaque but un plan choisi dans un menu. Dans la dernière étape, l'apprenant choisit des morceaux de code Pascal pour instantier les plans, et construit ainsi progressivement le programme en utilisant un éditeur syntaxique.

c) environnements d'apprentissage

"Pour l'initiation au concept de programmation sont souvent utilisés des environnements spécifiques, d'accès facile, n'imposant pas trop de contraintes syntaxiques et permettant de parvenir rapidement à des réalisation motivantes" ([Gra88]). C'est le cas des environnements **Logo** [Pap73] et **Micro-Prolog** ("Logic as a Computer Language for Children" [Enn81]).

Pour l'initiation au langage Prolog, on peut citer le logiciel d'enseignement **Apilog** ([BFW84]) qui est un produit commercialisé comprenant à la fois un didacticiel d'initiation au langage Prolog et un interpréteur Prolog permettant à l'apprenant d'écrire et de tester ses programmes.

Le projet Arcade ([PCP 88], [CGL 88]) est développé à Grenoble, à l'IMAG: le thème central de ce projet est celui d'un "laboratoire" (simulé) pour l'enseignement de la programmation. Ce laboratoire repose sur deux orientations pédagogiques fondamentales: liberté laissée à l'apprenant dans le choix et l'organisation des activités proposées, et participation fortement active à l'intérieur de celles-ci. *Manipulation*, *jeu* et *visualisation* y sont des modes de travail privilégiés. Les "salles" actuelles du laboratoire proposent à l'étudiant une exploration libre sur des thèmes tels que: tris, récursivité, listes, arbres binaires, spécification, pattern matching, etc ...

- *manipulation*: l'exemple du "Meccano de tri" ([Cag88]) concrétise cette idée de manipulation d'objets avant formalisation algorithmique. L'apprenant dispose d'outils permettant de réaliser des tris d'objets

représentés à l'écran par des cubes. Ces outils sont de divers niveaux: déplacer un cube vers une zone de travail; échanger deux cubes; rechercher le cube contenant la plus petite valeur; etc. L'apprenant peut choisir librement des outils parmi ceux qui lui sont proposés. Il doit ensuite travailler uniquement avec les outils qu'il a initialement choisis. Selon les choix faits, il devient possible (ou non) de faire, plus ou moins rapidement, des tris suivant tel ou tel algorithme. Notons que l'apprenant reste libre dans ses manipulations: on ne cherche ni à le guider vers une solution, ni à vérifier si ses manipulations sont conforme à un algorithme particulier.

- *visualisation*: l'exemple "Tris internes" ([LiP88]) permet l'observation de l'exécution des principaux algorithmes de tris internes. Le principe de la visualisation d'un tableau a été repris des travaux du projet Balsa ([BrS85]) précédemment cité: un élément du tableau est symbolisé par un trait proportionnel à la valeur de cet élément. Ainsi le tri est exécuté sur une représentation graphique du tableau. Le logiciel permet d'une part de commencer l'exécution d'un algorithme de tri en choisissant les données à trier, d'autre part de comparer des algorithmes de tri, à l'aide d'une évaluation du temps d'exécution et des nombres de comparaisons et d'affectations effectuées.

- *jeu*: le thème spécification, est illustré par un jeu de rôles: "Toi, moi et lui" ([GuP88], [Gue89]), pratiqué par trois joueurs sur un réseau de Macintosh. Ce jeu met en évidence, par la pratique, les comportements nécessaires au programmeur dans le travail qui le mène de l'énoncé du problème à l'une de ses solutions. Il s'agit de proposer à l'apprenant de mener à bien une sorte de projet, en mettant en valeur les comportements et en otant tous les détails de programmation. Le domaine choisi, pour concrétiser le projet, est celui de la bande dessinée. Le thème du jeu est la création d'une page de bande dessinée.

3.1.2. Contexte pédagogique du logiciel Idalgo

Notre expérimentation s'appuie sur un contexte d'enseignement de l'algorithmique et de la programmation, en début de second cycle universitaire à l'Université Joseph Fourier de Grenoble ([LPS83], [Sch84], [ScP88]).

Le cours d'algorithmique est organisé en *thèmes* qui en marque les étapes logiques. "Un thème est un ensemble cohérent de concepts; la composition d'un thème (les concepts) ainsi que la progression dans la présentation des thèmes résultent de choix pédagogiques fondamentaux" ([PeG 85]). L'ouvrage [ScP 88] traite des trois premiers thèmes de cette progression:

- *le thème 1* est une introduction aux structures algorithmiques élémentaires et à la problèmatique d'analyse.
- *le thème 2* est consacré aux structures itératives: construction d'algorithmes itératifs et traitement de séquences.
- *le thème 3* présente les techniques permettant de gérer un ensemble d'informations sous forme d'une séquence.

Les différents aspects d'un thème sont présentés concrétement au travers d'*exemples-types*. Ces exemples-types sont choisis de manière à n'aborder qu'une difficulté à la fois. Chaque exemple type caractérise une notion de base (par exemple: analyse par cas, présentation de la structure de suite, tableaux, ...), tout en faisant éventuellement appel à d'autres notions. L'ensemble des notions ainsi illustrées sont de nature variée: méthodes d'analyse, techniques de programmation, éléments de notation algorithmique et de langage de programmation, etc.

Un exemple type s'appuie toujours sur un énoncé de problème dont on veut une solution informatique. Le domaine de ce problème doit pouvoir être adapté au contexte culturel du public. A chaque exemple est associé un ensemble d'exercices de même type, renforçant le caractère modélisant de l'exemple de référence.

Pour fournir un cadre à la présentation, du passage de l'énoncé d'un problème à un algorithme suivant diverses phases d'analyse, les conventions adoptées dans le contexte d'enseignement, correspondent en partie à celles élaborées par le groupe AnnaGram de l'AFCET ([Gra86]). Le traitement d'un problème est ainsi rédigé sous forme d'une succession d'énoncés intermédiaires qui conduisent de l'énoncé initial à un algorithme. Chaque énoncé intermédiaire conclut une phase d'analyse et complète les énoncés précédents.

Pour notre expérience, nous nous centrons sur le deuxième thème de la progression pédagogique, consacré aux structures itératives et à la programmation par application de *schémas* dits de *traitement séquentiel* (nous donnons, en annexe ·, les notions abordées dans ce thème et des exemples de constructions d'algorithmes).

Nous montrons dans les figures qui suivent, qui traitent un même exemple (moyenne d'une suite de nombres), le paralléle possible entre la notion de *schéma* (figure 3.1) et la notion de *plan* (figure 3.2) utilisée dans plusieurs des logiciels d'enseignement de la programmation précédemment cités (Capra et Bridge où les plans servent de guides dans la construction, Proust où ils servent à la détection d'erreurs).

figure 3.1: application de schémas de traitement séquentiel ([Sch79])

PROBLEME: écrire un programme Pascal lisant des valeurs entières et les additionnant jusqu'à ce que la valeur 99999 soit lue. Ecrire la moyenne des nombres lus (la valeur 99999 non comprise).

ANALYSE:
- choix d'une classe de problème: il correspond à un parcours d'une séquence.
- définition de la séquence à traiter: c'est la séquence des valeurs lues.
- analyse du traitement: pour calculer la moyenne, on doit calculer la somme des entiers et la longueur de la séquence. Ces valeurs étant indépendantes, on peut les calculer simultanément lors d'un seul parcours de la séquence. On peut ensuite déduire la moyenne en distinguant le cas de la séquence vide.

Remarques de notation: Dans la définition de l'invariant et dans les définitions récurrentes qui vont suivrent, pg désigne la séquence d'éléments situés à gauche de l'élément courant, [] désigne la séquence vide et S.e désigne la séquence obtenue en ajoutant un élément e à droite (à la fin) d'une séquence S.

Soit **somme**(S) la somme des éléments d'une séquence S d'entiers, et **longueur**(S) sa longueur. On introduit un compteur **cpt** et un accumulateur **total**. Ces variables sont caractérisées par l'invariant: cpt = longueur(pg) et total = somme(pg). Les fonctions somme et longueur vérifient les relations:

longueur([]) = 0,   longueur(S.e) = longueur(S) + 1

somme([]) = 0,   somme(S.e) = somme(S) + e

On en déduit, l'initialisation du traitement: cpt <— 0 total <— 0 et le traitement de l'élément courant: cpt <— cpt + 1 total <— total + entcour

- application d'un schéma: le schéma à appliquer, correspond à la première forme du schéma de parcours d'une séquence marquée (avec traitement intégré de la séquence vide et du premier élément)

SCHEMA UTILISE:

Démarrer
Initialisation du traitement
tantque élément courant ≠ marque faire
    Traitement de l'élément courant
    Avancer
fintantque
Terminaison du traitement

INSTANTIATION DU SCHEMA:

Description de l'accés

Démarrer:
écrire('donnez un nombre')
lire(entcour)

Avancer:
écrire('donnez un nombre')
lire(entcour)

élément courant ≠ marque:
entcour ≠ 99999

Description du traitement

Initialisation du traitement
cpt <— 0
total <— 0

Terminaison du traitement
si cpt ≠ 0
    alors ecrire(total / cpt)
    sinon ecrire('pas d'entrées valides')
finsi

Traitement de l'élément courant
total <— total + entcour   cpt <— cpt + 1

ALGORITHME-INSTANCE OBTENU:

```
ecrire('donner un nombre')
lire(entcour)
cpt <— 0
total <— 0
tantque entcour ≠ 99999 faire

   total <— total + entcour
   cpt <— cpt + 1
   ecrire('donner un nombre')
   lire(entcour)
fintantque
si cpt ≠ 0
   alors  écrire( total / cpt )
   sinon  écrire('pas d'entrées valides')

finsi
```

PROGRAMME PASCAL

```
writeln('donner un nombre');
   read(entcour);
   cpt := 0;
   total := 0;
while entcour <> 99999 do
   begin
      total := total + entcour;
      cpt := cpt + 1;
   writeln('donner un nombre');
      read(entcour);
   end;
   if cpt < > 0
   then writeln( total / cpt )
   else
      writeln('pas d'entrées valides');
```

Remarques: nous avons utilisé des polices de caractères différentes, pour l'expression des parties concernant l'accés à la file ("en souligné"), et des parties concernant le traitement des éléments de la file ("en gras").

figure 3.2: analyse par buts, plans et instances de plans ([Sol 86])

Problem: write a program that will repeatedly read in and sum data until a final stopping value of 99999 is input. Output the average of the numbers read in.

Explanation:

		goal: *compute sum*	RUNNING TOTAL LOOP PLAN	goal: *realise stopping condition*	**SENTINELCONTROLLED RUNNING TOTAL LOOP PLAN**
	goal: *compute average*	goal: *compute count*	COUNTER LOOP PLAN	goal: *realize stopping condition*	SENTINEL CONTROLLED COUNTER LOOP PLAN
goal: *write program*	goal: *compute division*				DIVISION PLAN
				goal: *protect divide by zero*	**GUARD PLAN**
	goal: *output average*				*PRINT PLAN*

Program:

```
count := 0;
total := 0;
writeln(' input number');
read(new);
while new <> 99999 do
        begin
                        total := total + new;
                        count := count + 1;
                        writeln(' input number');
                        read(new);
        end;
if count > 0
        then
         begin
                        average := total / count;
                        writeln(average);
         end
        else
                        writeln('no valid inputs;
                        no average calculated');
```

Remarques: pour montrer la correspondance entre un plan utilisé dans l'analyse et son instance dans le programme, on utilise des polices de caractères différentes. Par exemples: "souligné" pour le SENTINEL-CONTROLLED COUNTER LOOP PLAN, "gras" pour le SENTINEL-CONTROLLED RUNNING-TOTAL LOOP PLAN, et "**gras souligné**" pour la fusion des deux).

3.2. Cahier des charges pédagogiques du logiciel Idalgo

Par rapport aux types de logiciels présentés en 3.1.1, le logiciel Idalgo est à la fois un environnement de programmation pour débutants ("style" Bridge) et également un logiciel informatif ("style" Satire).

3.2.1. Objectifs pédagogiques

L'objectif général de ce logiciel est de compléter l'enseignement d'algorithmique et de programmation présenté au paragraphe précédent, en offrant aux apprenants des ressouces E.A.O. On se place à un niveau d'initiation en milieu universitaire. Les principaux apports spécifiques attendus d'une activité E.A.O. dans ce domaine portent sur les points suivants:

- compréhension des divers niveaux de lecture d'un algorithme: comment *a-t-il été conçu,* comment *est-il exécuté,* comment *peut-il être traduit dans tel ou tel langage de programmation.* L'E.A.O. doit aider à l'analyse de la structure d'un programme par la visualisation de divers niveaux d'abstraction, de composants essentiels, des relations entre ces composants. Une bonne perception de ces mécanismes débouche sur une meilleure capacité d'abstraction: ceci est essentiel pour l'assimilation et la pratique de méthodes de programmation;
- compréhension des principes de modélisation informatique: notion de spécification, de structuration et de représentation des actions et des informations;
- compréhension des mécanismes dynamiques qui sous-tendent une description algorithmique: contrôle de l'exécution et schémas de composition fondamentaux; modification de l'état des variables et structures de données. L'E.A.O. est particulièrement adapté à la visualisation de phénomènes dynamiques qu'il est par ailleurs difficile de présenter par des méthodes traditionnelles.

3.2.2. Activités pédagogiques

L'enseignement que doit soutenir le logiciel est fondé sur un ensemble d'*exemples types*. Chaque exemple type comporte un énoncé, un ou plusieurs algorithmes solution et les programmes associés. On pourra prévoir, pour un exemple type donné, plusieurs énoncés de départ pris dans des domaines différents (exemples numériques, traitement de textes, dessins de figures, ...), permettant à l'apprenant de travailler dans un contexte qui lui est adapté. De même, on pourra l'illustrer par plusieurs langages de programmation.

A chaque exemple type sont associées plusieurs *activités*, répondant aux objectifs généraux décrits plus haut. Les activités illustrent le concept présenté dans un exemple type sous plusieurs angles et s'attachent à la fois à

isoler les notions propres à l'exemple, et à les replacer dans un contexte général de construction de programmes (pour faire la liaison entre divers concepts). Elles sont définies sur la base des questions générales suivantes:
- comment analyse-t-on un énoncé pour produire un algorithme ? Paradigmes d'analyse propres à l'exemple, étapes d'analyse, mise en évidence des choix: *activité de construction en mode écriture*,
- comment lit-on un algorithme ? Repérage des composants, mise en évidence de la structure (niveaux d'abstraction induits par le texte de l'algorithme): *activité de construction en mode lecture*,
- comment l'algorithme est-il exécuté ? Interprétation opératoire de chaque composant, étude de comportement sur plusieurs données significatives: *activité de visualisation de l'exécution*,
- comment produit-on, à partir de l'algorithme, un programme dans un langage donné ? Règles de traduction systématique, transformations rendues nécessaires par le langage: *activité de traduction*.

3.2.3 Modèles de dialogues des activités

Nous présentons les modèles de dialogues des différentes activités. Ils servent à fixer le cadre de présentation de l'activité, indépendamment de l'exemple traité. Pour chaque activité, nous rappellons son objectif pédagogique, nous la présentons, nous donnons un modèle de dialogue associé, et un exemple d'instantiation du modèle dans le cas de l'exemple type *compter les 'le'*: "écrire un algorithme qui affiche le nombre d'occurences du couple 'le' dans un texte terminé par un point".

a] Dialogues de construction d'un algorithme

Objectif pédagogique

Il s'agit de faire travailler l'apprenant sur les étapes successives de la construction d'un algorithme à partir de l'énoncé d'un problème. On distingue deux modes: *mode écriture*, dont l'objectif est de fournir un guide

dans la construction d'un algorithme en permettant de choisir puis d'instantier un schéma d'algorithme; *mode lecture*, où l'on montre les étapes de la construction d'un algorithme et la structure d'un algorithme par rapport à son analyse.

Présentation de l'activité

Pour un énoncé donné on trouve les étapes suivantes: choix d'un modèle d'accès séquentiel et choix d'un schéma de traitement (Ecran 1), description de la machine séquentielle d'accés et description de la machine de traitement (Ecran 2), synthèse de l'algorithme (Ecran 3).

Un modèle de dialogue:

Ecran 1

Activité de construction d'un algorithme

Enoncé de l'exercice

TEXTE DE L'ENONCE DE L'EXERCICE

Choix d'un type d'accés:

- ■ machine de type 1
- □ machine de type 2

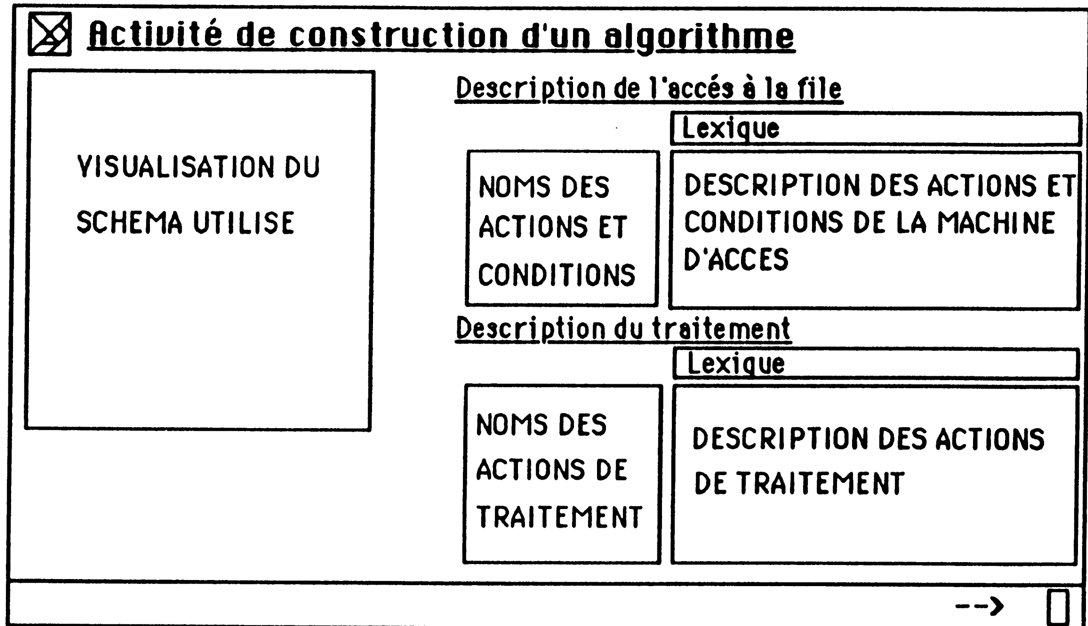
Choix d'un schéma:

- ■ parcours de file
- □ parcours de sous-file définie par inclusion
- □ parcours de sous-file définie par exclusion
- □ recherche dans une file

--> □

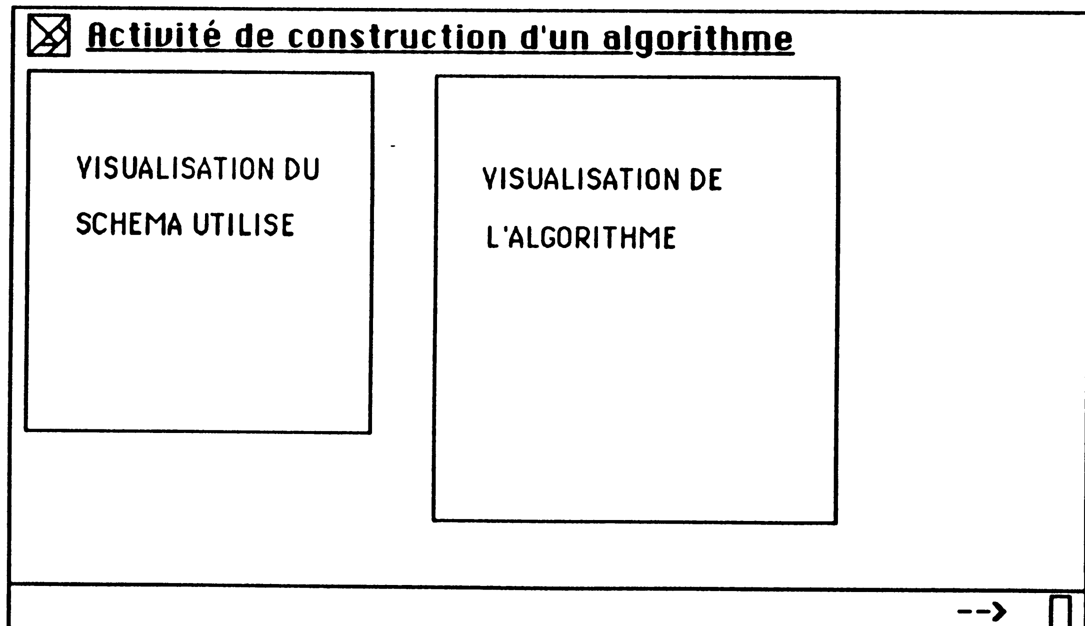
- en mode écriture: l'apprenant choisit pour un énoncé donné, un type de machine et un schéma qu'il pense correspondre au problème posé,
- en mode lecture: l'apprenant voit que pour tel problème, on peut l'analyser en utilisant tel type de machine et tel schéma. Il pourra éventuellement disposer de plusieurs solutions pour un même énoncé.

Ecran 2



- en mode écriture: l'apprenant instantie le schéma, en fournissant la description de l'accés et du traitement de la file,
- en mode lecture: il consulte ces descriptions.

Ecran 3



- en mode écriture et en mode lecture: l'apprenant obtient l'algorithme instance du schéma utilisé.

Exemple d'instantiation du modèle:

L'écran 1 de cet exemple correspond au choix d'un accès séquentiel de type 1 et au choix d'un schéma de parcours de file.

Ecran 2

Activité de construction d'un algorithme

```
nommer logique fdf;
initp
dem
tantque non fdf faire
   visiter
   av
ftantque
termp
```

Description de l'accés à la file

```
                nommer caractere carcour,cp;
dem:            cp<- '_'
                demcaractere
av:             cp<- carcour
                avcaractere
non fdf:        carcour ≠ '.'
```

Description du traitement

```
                nommer entier cpt;
initp:          cpt<- 0
termp:          ecrire( cpt )
visiter:        si (cp = 'l') et (carcour = 'e')
                   alors    cpt<- cpt + 1
                fsi
```

-->

Remarque: la file décrite, est la file des couples de caractères du texte.

Ecran 3

Activité de construction d'un algorithme

```
nommer logique fdf;
initp
dem
tantque non fdf faire
   visiter
   av
ftantque
termp
```

```
nommer caractere carcour,cp;
nommer entier cpt;
cpt<- 0
cp<- '_'
demcaractere
tantque carcour ≠ '.' faire
   si (cp = 'l') et (carcour = 'e')
      alors    cpt<- cpt + 1
   fsi
   cp<- carcour
   avcaractere
ftantque
ecrire( cpt )
```

-->

b) dialogues de visualisation de l'exécution d'un algorithme

Objectif pédagogique

Dans l'activité précédente, l'algorithme sert de support à une explicitation de l'analyse qui y a conduit. On décrit le processus dynamique lors de son exécution. On donne à l'apprenant l'interprétation de l'algorithme en termes d'exécution.

Présentation de l'activité

L'activité consiste à visualiser, sur un jeu d'essai donné par l'apprenant, l'exécution de l'algorithme en montrant (Ecrans 2 à n) l'action en cours d'exécution, l'évolution de l'état des variables liées à l'application de l'action, éventuellement une simulation graphique de l'action en cours d'exécution (exemples: actions **dem-caractère** et **av-caractère** qui font avancer un ruban de caractères dans une fenêtre d'une machine caractères).

Un modèle de dialogue

Ecran 1

Ecrans 2 à n

Activité d'exécution d'un algorithme

ALGORITHME

Construction algorithmique en cours d'exécution

Etat des variables

ETAT INTERMEDIAIRE

Variable modifiée

Entrées

Sorties

-->

Remarque: les erreurs éventuelles commises par l'apprenant sur ses algorithmes, sont détectées à l'exécution (par exemple, **avcaractère** sans **demcaractère** préalable, division par zéro...) et provoquent l'affichage d'un message commentant l'erreur dans la zone de communication au bas de l'écran.

Ecran final

Activité d'exécution d'un algorithme

ALGORITHME

Etat des variables

ETAT FINAL

Entrées

Sorties

-->

Exemple d'instantiation du modèle:

Ecran 1

Activité d'exécution d'un algorithme

```
nommer caractere carcour,cp
nommer entier cpt
cpt <- 0
cp <- '_'
demcaractere
tantque carcour ≠ '.' faire
  si (cp = 'l') et (carcour = 'e')
     alors cpt <- cpt + 1
  fsi
  cp <- carcour
  avcaractere
ftantque
ecrire( cpt )
```

Etat des variables

	carcour
	cp
	cpt

Entrées

Sorties

-->

Ecran n

Activité d'exécution d'un algorithme

```
nommer caractere carcour,cp
nommer entier cpt
cpt <- 0
cp <- '_'
demcaractere
tantque carcour ≠ '.' faire
  si (cp = 'l') et (carcour = 'e')
     alors cpt <- cpt + 1
  fsi
  cp <- carcour
  avcaractere
ftantque
ecrire( cpt )
```

Etat des variables

l	**carcour**
_	**cp**
0	**cpt**

Entrées

laleles.

Sorties

-->

Remarque: le jeu d'essai donné par l'apprenant, est ici le texte "laleles.". L'action demcaractere provoque le positionnement sur le premier caractère de ce texte: le caractère 'l'.

Ecran n':

Activité d'exécution d'un algorithme

```
nommer caractere carcour,cp
nommer entier cpt
cpt <- 0
cp <- '_'
demcaractere
tantque carcour ≠ '.' faire
  si (cp = 'l') et (carcour = 'e')
    alors cpt <- cpt + 1
  fsi
  cp <- carcour
  avcaractere
ftantque
ecrire( cpt)
```

Etat des variables

e	carcour
l	cp
1	cpt

Entrées

laleles.

Sorties

-->

Ecran final

Activité d'exécution d'un algorithme

```
nommer caractere carcour,cp
nommer entier cpt
cpt <- 0
cp <- '_'
demcaractere
tantque carcour ≠ '.' faire
  si (cp = 'l') et (carcour = 'e')
    alors cpt <- cpt + 1
  fsi
  cp <- carcour
  avcaractere
ftantque
ecrire( cpt)
```

Etat des variables

.	carcour
s	cp
2	cpt

Entrées

laleles.

Sorties

2

-->

c) dialogues de traduction d'un algorithme

Objectif pédagogique

Ces dialogues sont consacrés à la visualisation du processus de traduction d'un algorithme vers un programme dans un langage de programmation donné. Nous souhaitons montrer l'aspect systématique du

passage d'un algorithme à un programme, par l'application de règles de traduction. L'écriture du programme passe toujours par l'écriture d'un algorithme dans la notation algorithmique.

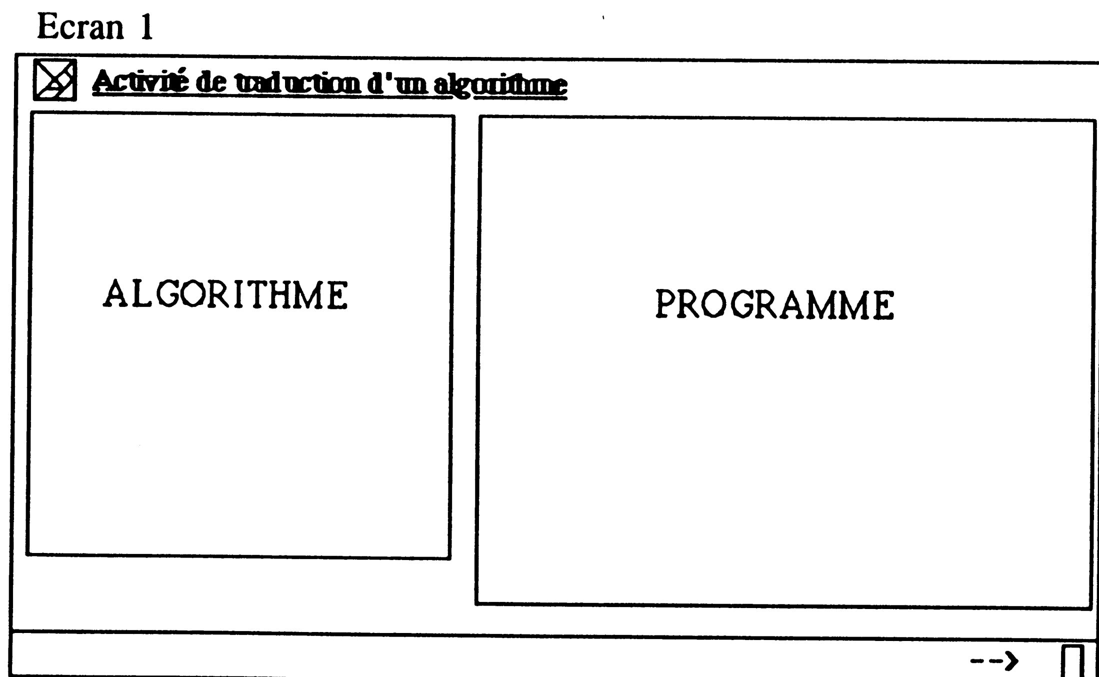
Cette traduction permet la concrétisation par un changement de formalisme de concepts manipulés abstraitement par ailleurs. C'est pourquoi, plutôt que de se limiter à un seul langage cible, il nous semble préférable, d'aborder plusieurs langages de programmation de niveaux différents. Nous avons choisi Pascal et Basic.

Présentation de l'activité

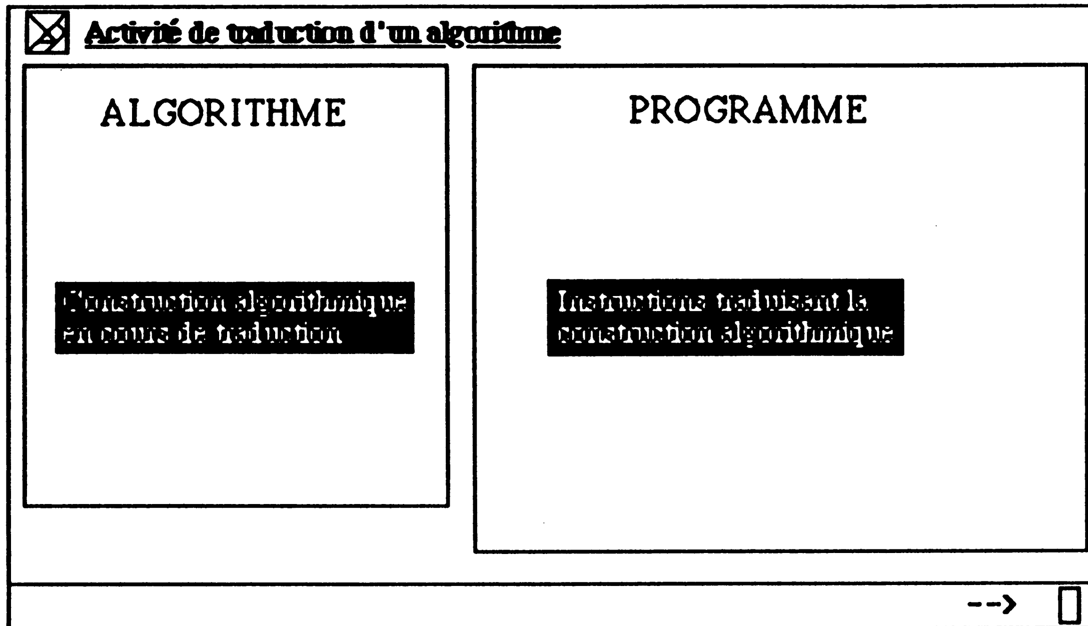
Nous visualisons le passage d'un algorithme à un programme (Ecran 1) par traduction systématique des éléments de la notation algorithmique, en les éléments correspondant du langage cible. Les écrans 2 à n correspondent aux différentes étapes du processus de traduction. Chaque étape montre l'application d'une règle de traduction correspondant à la construction algorithmique à traduire.

Un modèle de dialogue:

Ecran 1

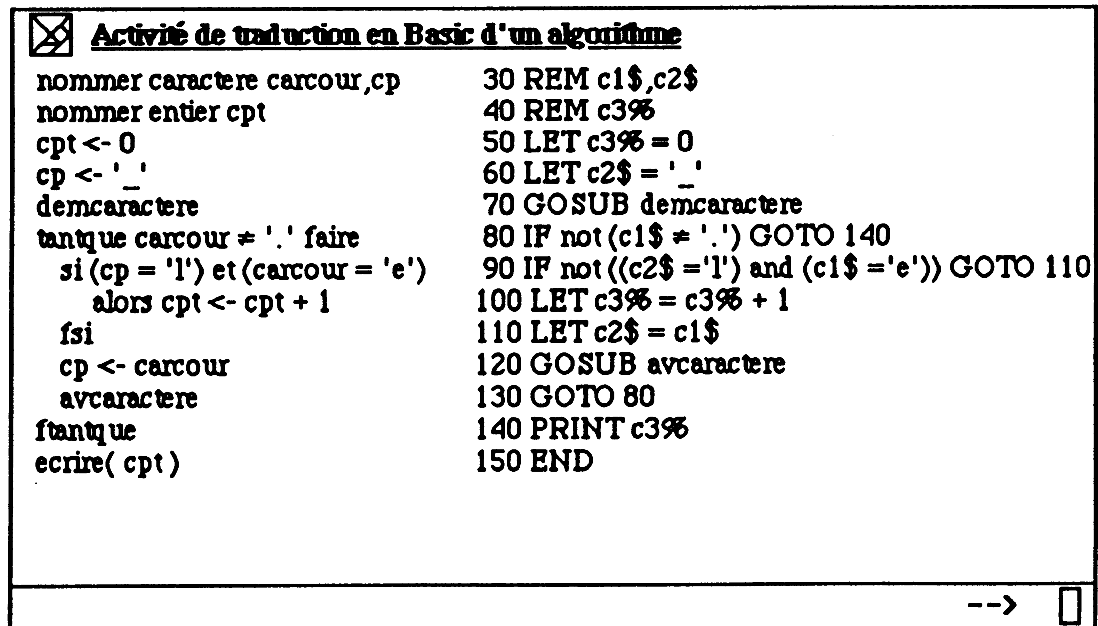


Ecrans 2 à n



Exemple d'instantiation du modèle: version Basic

Ecran 1



Ecran n

Activité de traduction en Basic d'un algorithme

```
nommer caractere carcour,cp          30 REM c1$,c2$
nommer entier cpt                    40 REM c3%
cpt <- 0                             50 LET c3% = 0
cp <- '_'                            60 LET c2$ = '_'
demcaractere                         70 GOSUB demcaractere
tantque carcour ≠ '.' faire          80 IF not (c1$ ≠ '.') GOTO 140
  si (cp = 'l') et (carcour = 'e')   90 IF not ((c2$ ='l') and (c1$ ='e')) GOTO 110
    alors cpt <- cpt + 1             100 LET c3% = c3% + 1
  fsi                                110 LET c2$ = c1$
  cp <- carcour                      120 GOSUB avcaractere
  avcaractere                        130 GOTO 80
ftantque                             140 PRINT c3%
ecrire( cpt )                        150 END
```

-->

Ecran n'

Activité de traduction en Basic d'un algorithme

```
nommer caractere carcour,cp          30 REM c1$,c2$
nommer entier cpt                    40 REM c3%
cpt <- 0                             50 LET c3% = 0
cp <- '_'                            60 LET c2$ = '_'
demcaractere                         70 GOSUB demcaractere
tantque carcour ≠ '.' faire          80 IF not (c1$ ≠ '.') GOTO 140
  si (cp = 'l') et (carcour = 'e')   90 IF not ((c2$ ='l') and (c1$ ='e')) GOTO 110
    alors cpt <- cpt + 1             100 LET c3% = c3% + 1
  fsi                                110 LET c2$ = c1$
  cp <- carcour                      120 GOSUB avcaractere
  avcaractere                        130 GOTO 80
ftantque                             140 PRINT c3%
ecrire( cpt )                        150 END
```

-->

Exemple d'instantiation du modèle: version Pascal

Ecran n

Activité de traduction en Pascal d'un algorithme

```
nommer caractere carcour,cp
nommer entier cpt
cpt <- 0
cp <- '_'
demcaractere
tantque carcour ≠ '.' faire
  si (cp = 'l') et (carcour = 'e')
    alors cpt <- cpt + 1
  fsi
  cp <- carcour
  avcaractere
ftantque
ecrire( cpt )
```

```
var carcour,cp : char;
    cpt : integer;
begin
cpt := 0
cp := '_'
demcaractere
while carcour ≠ '.' do
  begin
  if (cp = 'l') and (carcour = 'e')
    then cpt := cpt + 1;
  cp := carcour;
  avcaractere;
  end;
write(cpt);
end.
```

-->

d) aides accessibles pour les différentes activités:

Pour l'ensemble de ces activités l'apprenant dispose d'aides concernant les différentes constructions algorithmiques: forme syntaxique, règles d'interprétation sous forme de schémas d'exécution, et règles de traduction en Pascal ou en Basic. Par exemple, pour la traduction en Basic de la construction itérative "tantque", on affiche une règle sous la forme suivante:

```
tantque CONDITION faire      n°x  IF NOT(CONDITION') GOTO n°y
    ACTIONS                        ACTIONS'
ftantque                           GOTO n°x
                             n°y ...
```

Remarques: n°x et n°y représentent des numéros de lignes en Basic. CONDITION' et ACTIONS' correspondent à la traduction Basic de CONDITION et ACTIONS.

3.2.4. Variations dans la présentation des objets et des processus

Le logiciel Idalgo visualise
- des *objets*: algorithmes, instances, schémas, règles concernant la notation algorithmique (règles syntaxiques, règles de présentation, de vérification de types, d'interprétation, de traduction);
- des *mises en correspondance d'objets*: entre un schéma et une instance de schéma, une instance et un algorithme, un algorithme et un programme;
- et des *processus*: construction, interprétation, traduction.

Nous donnons quelques exemples montrant la variété des formes de *présentations statiques* des objets et des mises en correspondance des objets manipulés dans le logiciel Idalgo, sachant qu'ils se doublent d'une variété de *présentations dynamiques* des processus.

a) présentations de nature syntaxique d'un algorithme

Ce type de présentation met en évidence la structure syntaxique de l'algorithme en dégageant les constructions algorithmiques utilisées. Les formes de visualisation de cette structure sont multiples. Nous en présentons ici trois grandes formes et trois exemples de superposition de plusieurs de ces formes puis nous parlons des degrés de liberté de ces présentations.

Une première forme consiste à utiliser l'indentation:

exemple 1

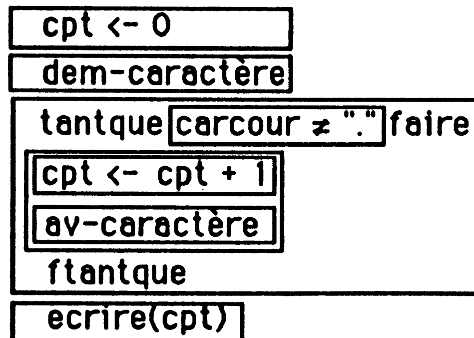
```
cpt <- 0
dem-caractère
tantque carcour ≠ "." faire
    cpt <- cpt + 1
    av-caractère
ftantque
ecrire(cpt)
```

exemple 2

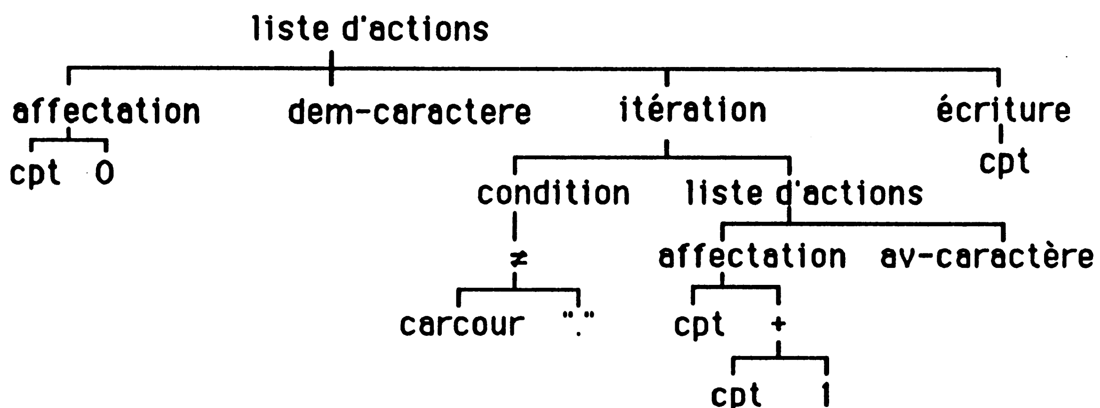
```
cpt <- 0
dem-caractère
  tantque carcour ≠ "." :
    cpt <- cpt + 1
    av-caractère
ecrire(cpt)
```

Remarque: dans l'exemple 2, on voit que l'indentation peut être utilisé pour alléger les conventions de notation, ici en supprimant des mots clés faire et ftantque. Cette présentation est utilisée dans ([Gra86]).

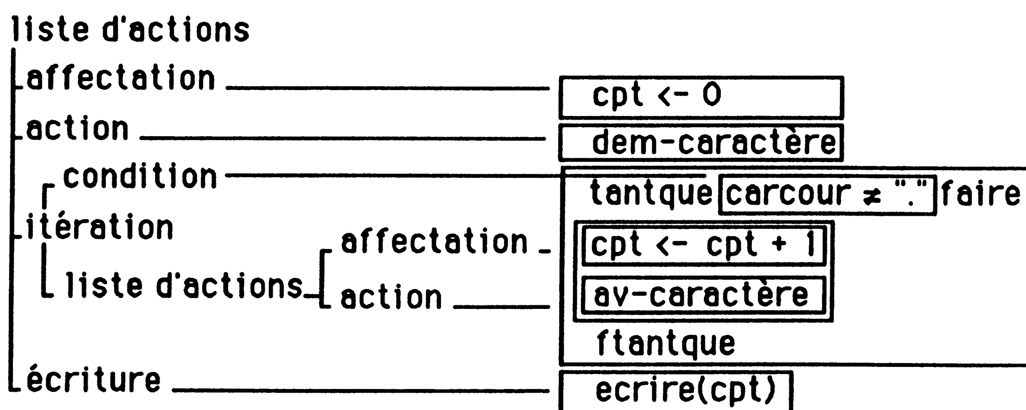
Une deuxième forme de présentation consiste à utiliser une représentation par emboîtements successifs, par exemple:



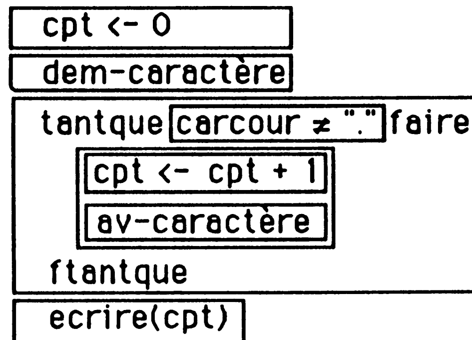
Une troisième forme correspond à une représentation en arbre, par exemple:



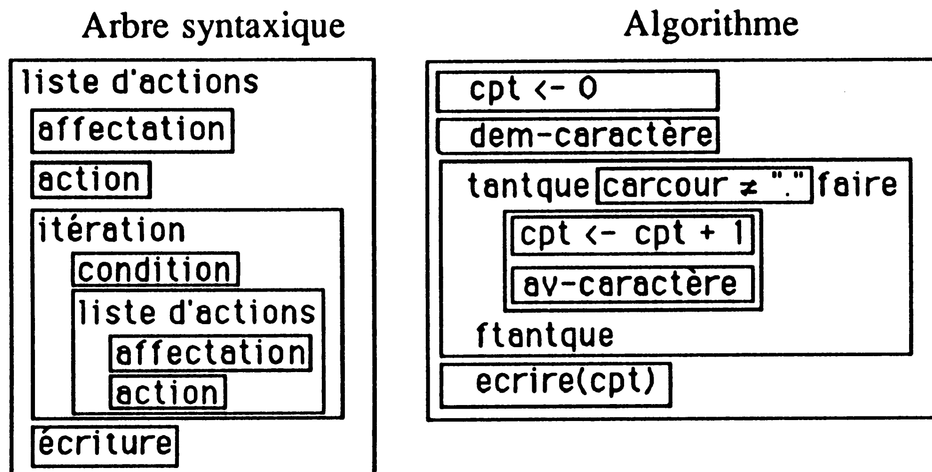
Nous montrons maintenant un exemple de superposition de la deuxième et de la troisième forme citées ci-dessus:



Une autre manière de superposer les deux premières formes consiste à renforcer la présentation de la structuration syntaxique donnée par l'indentation, en encadrant les différents blocs, par exemple:



Enfin, on peut combiner une présentation sous forme d'arbre syntaxique et la mise en évidence de blocs:



La correspondance entre l'algorithme et l'arbre syntaxique associé est mise en évidence par les boites successives, d'autre part, nous avons aussi indenté l'arbre syntaxique.

b) degrés de liberté de ces présentations

Ces différentes présentations sont paramétrables et pour chaque présentation, nous n'avons montré qu'une instance possible des paramètres de la présentation.

Reprenons l'exemple 1 de la première forme de présentation en montrant que beaucoup de choix ont été faits.

```
cpt <- 0
dem-caractère
tantque carcour ≠ "." faire
        cpt <- cpt + 1
        av-caractère
ftantque
écrire(cpt)
```

Les choix effectués concernent le pas d'indentation, la localisation des mots-clès, la langue utilisée (français, anglais, espagnol, ...), les attributs d'aspect (souligné, double intensité, ...), le nombre d'instructions par ligne, l'existence d'espaces avant et aprés le symbole d'affectation ou à l'intérieur des expressions arithmétiques, etc.

Dans l'exemple 2 de la deuxième forme, un des choix possibles est de fermer une boite à la colonne qui suit le dernier caractère de l'instruction ou à la colonne qui suit le dernier caractère de l'instruction la plus longue du même bloc.

choix 1

```
cpt <- 0
dem-caractère
tantque [carcour ≠ "."] faire
    cpt <- cpt + 1
    av-caractère
ftantque
ecrire(cpt)
```

choix 2

```
cpt <- 0
dem-caractère
tantque [carcour ≠ "."] faire
    cpt <- cpt + 1
    av-caractère
ftantque
ecrire(cpt)
```

c) présentations d'un algorithme instance d'un schéma

Les différentes formes de présentations d'un algorithme que nous allons indiquer ne sont applicables que si l'algorithme a été construit en appliquant les schémas de traitement séquentiel. Ces présentations montrent

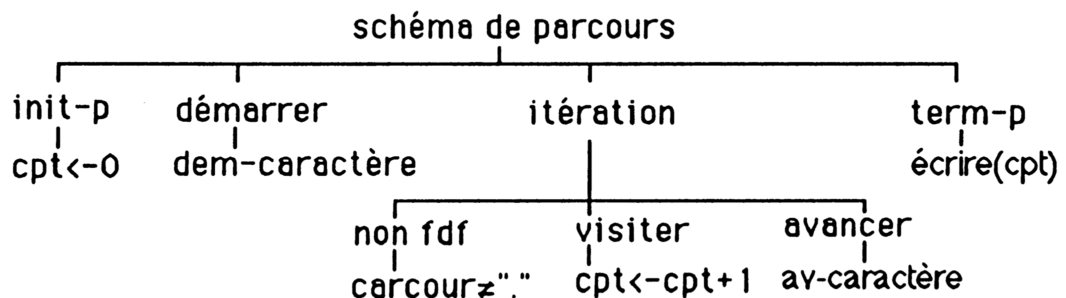
la structuration de l'algorithme par rapport au schéma utilisé pour le construire. Elles mettent en évidence les primitives d'accés à la file et celles de traitement utilisées.

Une première forme de présentation consiste à visualiser de façons différentes les primitives d'accés et celles de traitement:

cpt <- 0
DEM-CARACTERE
tantque CARCOUR ≠ "." faire
cpt <- cpt + 1
AV-CARACTERE
ftantque
ecrire(cpt)

légende:
Les primitives de traitement sont en minuscules soulignées.
Les primitives d'accés sont écrites en MAJUSCULES.

Une deuxième forme correspond à une représentation en arbre:



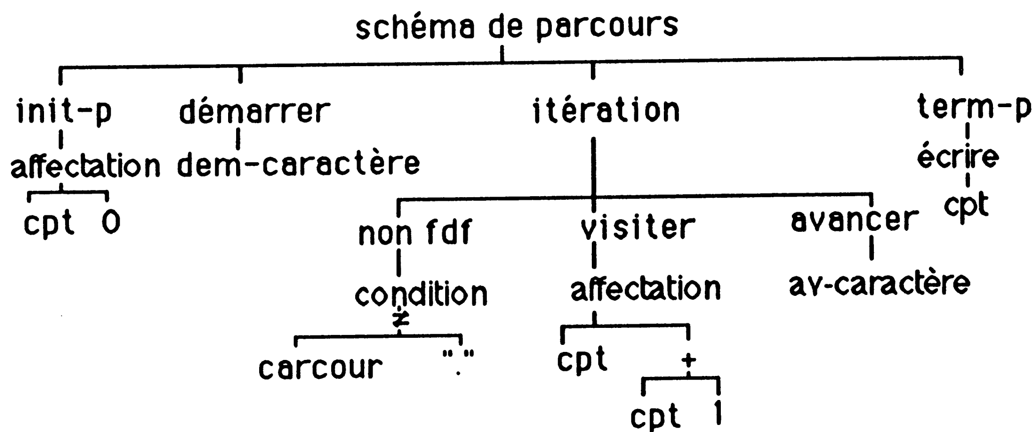
D'autres formes de présentation sont bien évidemment possibles.

d] Superposition des deux types de présentations précédents

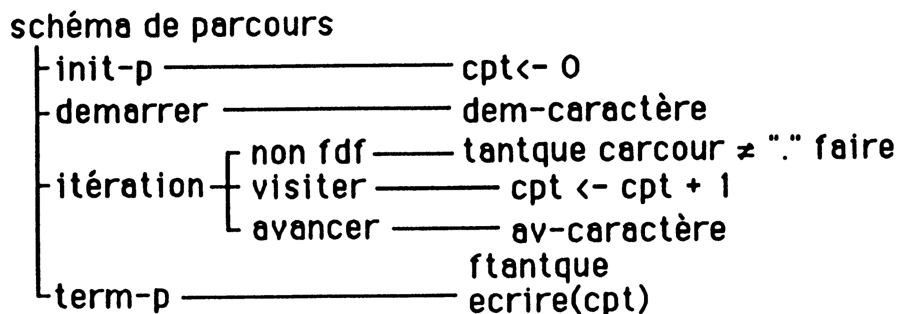
Il est possible de présenter simultanément les deux types de présentations. On peut par exemple, indenter les algorithmes et mettre en évidence les actions d'accès séquentiel et de traitement.

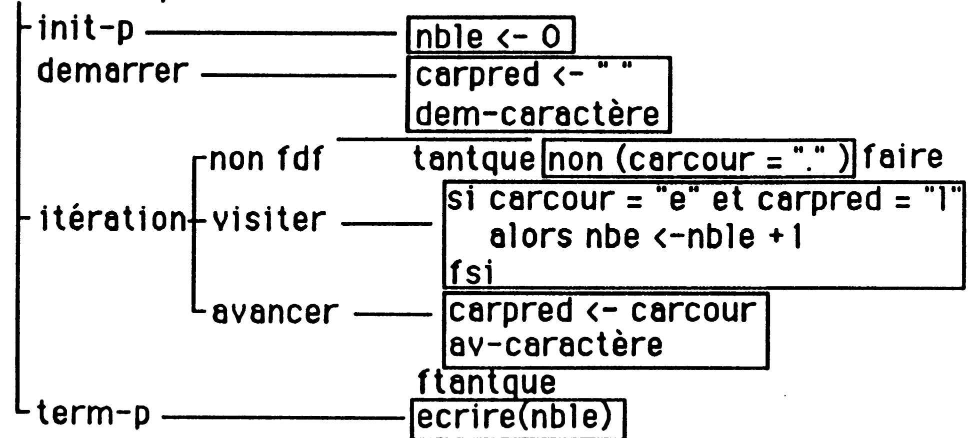
<u>cpt <- 0</u>
DEM-CARACTERE
tantque CARCOUR ≠ "." faire
    <u>cpt <- cpt + 1</u>
    AV-CARACTERE
ftantque
<u>écrire(cpt)</u>

On peut aussi avoir recours à un arbre abstrait qui met en évidence le schéma utilisé:

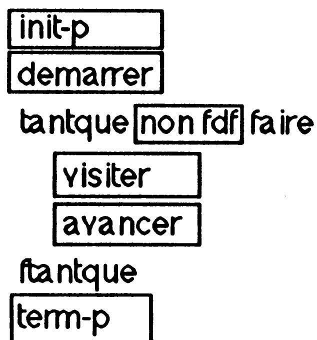


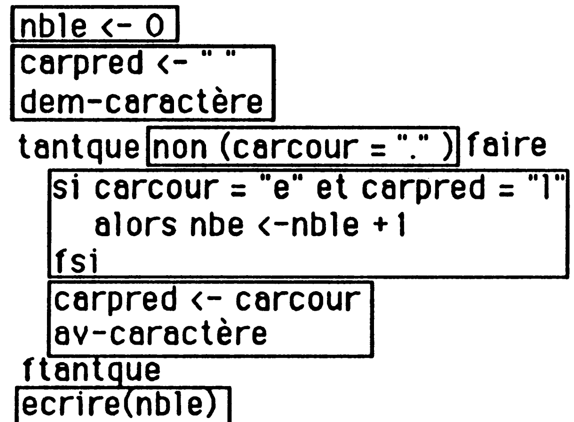
Dans les deux exemples qui suivent, l'algorithme est présenté sous forme indentée et le schéma sous forme d'arbre





On peut séparer les présentations du schéma et de l'algorithme:



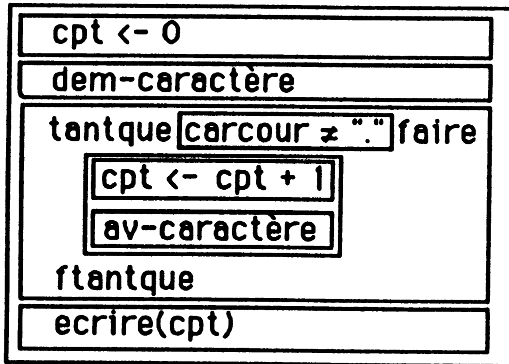


e) présentations de la correspondance entre un algorithme et un programme

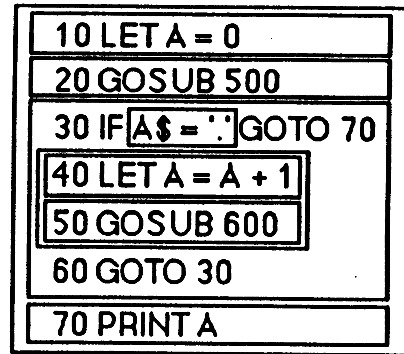
On montre la relation entre un algorithme et un programme obtenu par traduction de l'algorithme dans un langage de programmation.

Une première forme de présentation consiste à mettre en évidence la correspondance entre l'algorithme et le programme par des boites successives, par exemple:

Algorithme

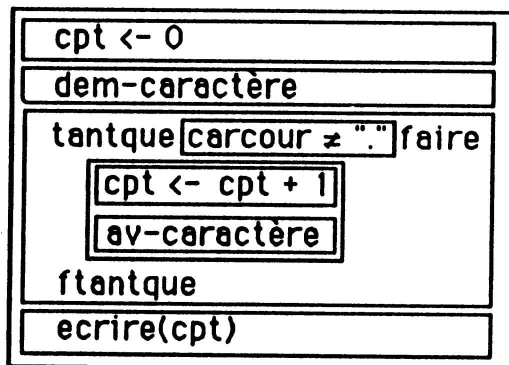


Programme Basic

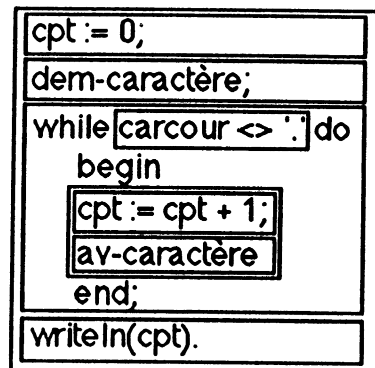


Remarque: cette présentation fait ressortir dans le programme les instructions de branchement (IF GOTO n° de ligne et GOTO n° de ligne)

Algorithme



Programme Pascal



Pour ces deux exemples, on voit que l'on a utilisé des règles de présentation des programmes: présentation du programme Basic avec ses numéros de lignes et sans indentation, présentation indentée du programme Pascal.

3.3. Conclusion

Rappelons que la conception du logiciel Idalgo fait suite à différentes maquettes de logiciels pour l'enseignement de l'algorithmique, préalablement réalisées. Ces maquettes ([Zam84], [Agu85], [Lie85], [Gue85]) ont été developpées sous différents systèmes de production: Mosaïque ([Ada83]), Voyelles ([Luc83]), Diane-Arlequin ([Cas84]), Diane-Editeur fonctionnel ([Eur85]).

En matière de production, le bilan de ces divers développements, a montré que ces différents systèmes étaient mal adaptés à la production d'un logiciel explorant les différentes activités souhaitées. En particulier, ils ne permettaient pas de manipuler simplement la structure des algorithmes, ni de décrire les activités comme des modèles de dialogues instantiables par l'algorithme présenté dans l'activité.

C'est dans cette optique, que nous avons travaillé sur une spécialisation du système Eaolog dans le domaine de l'enseignement de l'algorithmique et de la programmation, en ajoutant différents niveaux d'éditeurs traitant les objets du domaine: constructions algorithmiques; règles de présentation, d'interprétation, de traduction, etc; algorithmes, instances, schémas. Ce nouveau système Tangram , et l'approche de production associée, sont décrits au chapitre IV, où nous illustrons notre démarche de production sur l'exemple de la production du logiciel Idalgo.

Chapitre IV

PRODUCTION DU LOGICIEL IDALGO: LE SYSTEME TANGRAM

Le logiciel Idalgo est spécifié en termes d'objets pédagogiques, notation algorithmique, schémas, algorithmes, langages de programmation... Sa réalisation nécessite de définir les diverses actions associées à ces objets, en tenant compte des besoins spécifiques des divers intervenants et notamment de l'enseignant utilisateur.

La manipulation des objets s'appuie sur des moyens d'expression adéquats: l'auteur spécifie la notation algorithmique et les modèles des activités qu'il veut mettre à disposition dans le logiciel. L'enseignant utilisateur instancie ces modèles et adapte la notation algorithmique qu'il utilise lui-même pour donner les exemples spécifiques d'algorithmes sur lesquels doit travailler l'apprenant. Dans le cadre des dialogues ainsi définis, l'apprenant manipule des algorithmes au travers de la notation algorithmique ou de langages de programmation.

C'est ainsi que pour produire le logiciel Idalgo, nous avons choisi de réaliser une couche logicielle au dessus de Eaolog permettant de manipuler ces objets pédagogiques: le système Tangram (**T**raitement **a**ssisté de **n**otations spécifiées par des **gram**maires). En fait ce système peut être réutilisé dans d'autres contextes d'enseignement où les activités proposées s'appuient sur des objets très structurés, la maîtrise de ces structures jouant un rôle important dans l'apprentissage concerné.

Nous décrivons cette approche de production en précisant les besoins des différents intervenants, les fonctionnalités et l'architecture du système

de production, et les moyens de description des connaissances (§4.1). Nous présentons ensuite le système Tangram et son utilisation pour la production du logiciel Idalgo. Nous détaillons les possibilités d'expression et d'édition des objets pédagogiques et des modèles de dialogues (§ 4.2). Enfin, nous discutons du logiciel ainsi produit et d'extensions possibles (§ 4.3).

4.1. Approche de production

L'approche classique de production de didacticiels (§1.2) incite à penser le logiciel à produire, en termes d'une succession de dialogues et non en termes d'activités auxquelles on associe ensuite des modèles de dialogues. Cette démarche provoque de sérieux problèmes en ce qui concerne:
- la *cohérence globale* du didacticiel à produire: uniformisation des règles de présentation de l'algorithme, d'interprétation des constructions de la notation algorithmique, de traduction vers les langages de programmation...
- la *cohérence locale* des dialogues: compatibilité entre l'algorithme donné dans une activité de lecture et le schéma de référence dans une activité de construction; correction syntaxique et sémantique de l'algorithme présenté; adéquation entre l'algorithme et les programmes Basic ou Pascal donnés dans une activité de traduction; ...
- la *masse de programmes à produire*: par exemple pour la production d'un logiciel proposant les quatre activités sur vingt algorithmes différents, il faut réaliser quatre-vingts dialogues.
- l'impact des *modifications* ponctuelles sur l'ensemble: par exemple, un changement concernant une construction algorithmique ou sa traduction dans un langage doit être reportée sur l'ensemble des dialogues concernés.

Tenant compte de cette expérience, il est apparu nécessaire de définir un environnement de production de logiciels adapté aux objets pédagogiques propres à l'enseignement de l'algorithmique et de la

programmation, et aux modèles d'activités spécifiées au chapitre 3.

Dans un premier temps, nous avons *analysé les besoins* des différents intervenants du processus de production, et mis en évidence *les objets du domaine* et *les objets E.A.O.* manipulés par chacun d'eux.

Nous en avons déduit les *fonctionnalités et l'architecture d'un système production*, adapté à ces différents intervenants et à ce domaine d'application, construit autour du système Eaolog fournissant des fonctionnalités plus générales de l'E.A.O. Chacun des utilisateurs a ses besoins propres et doit pouvoir intervenir dans l'environnement en fonction de son niveau de compétence.

Enfin, nous avons étudié la manière d'exprimer les connaissances sur le domaine en référence aux méthodes de spécification formelle des langages. Nous avons ainsi poursuivi notre expérimentation sur l'apport de Prolog dans le domaine de l'EAO.

4.1.1. Analyse des besoins des différents intervenants

Lors de la phase de création, *l'enseignant-auteur* doit donner au système les connaissances sur la notation algorithmique: lexicographie, syntaxe, règles de présentation, sémantique opérationnelle, sémantique statique, traduction vers Basic et Pascal. Il doit aussi formuler les différentes activités qu'il souhaite proposer à l'apprenant. L'environnement doit lui fournir des moyens d'expression adaptés à la nature des objets manipulés et doit se prêter aux diverses phases de production, notamment la validation et les enrichissements progressifs par ajout de nouvelles connaissances.

L'*enseignant-utilisateur* doit pouvoir s'approprier le logiciel et le personnaliser. Par exemple en jouant sur la forme de visualisation des algorithmes et des processus ou par le choix de la notation algorithmique utilisée (sur l'ensemble des constructions mises à disposition par l'auteur) et des constructions algorithmiques accessibles à un instant donné. Cette

appropriation concerne donc d'une part, la notation algorithmique utilisée et les exemples de schémas, d'instances de schémas et d'algorithmes; d'autre part la forme de présentation des activités. Il doit donc pouvoir facilement modifier la notation algorithmique spécifiée par l'enseignant-auteur, modifier la présentation des activités et enrichir des bibliothèques de schémas, d'instance et d'algorithmes.

L'*apprenant* doit pouvoir créer ses propres bibliothèques de schémas, d'instances et d'algorithmes, qu'il doit pouvoir utiliser au travers des différentes activités, au même titre que ceux préparés par les enseignants. Il doit pouvoir consulter les informations concernant la notation algorithmique.

4.1.2. Fonctionnalités et architecture du système de production

Le système fournit des environnements de travail, comportant des moyens d'expression et des outils d'édition, adaptés aux trois classes d'utilisateurs. L'environnement enseignant-auteur permet l'expression et l'édition des connaissances sur la notation algorithmique (lexicographie, syntaxe, et sémantiques) et des modèles de dialogues des activités. L'environnement enseignant-utilisateur doit permet l'appropriation de la notation algorithmique et des modèles de dialogues, et l'expression et l'édition des exemples écrits dans la notation algorithmique. L'environnement apprenant permet la consultation des connaissances sur la notation algorithmique et l'exploitation de ces connaissances sur les exemples préparés par les enseignants ou donnés par l'apprenant lui-même.

L'*architecture des logiciels à produire* (figure 4.1) est proche de celle d'un système d'E.I.A.O. expert du domaine (§1.1.5, figure 1.2).

L'*architecture du système de production* (figure 4.2) permet à chacun des utilisateurs d'avoir accés, via des éditeurs spécialisés, aux différentes bases du système.

figure 4.1: architecture des logiciels à produire

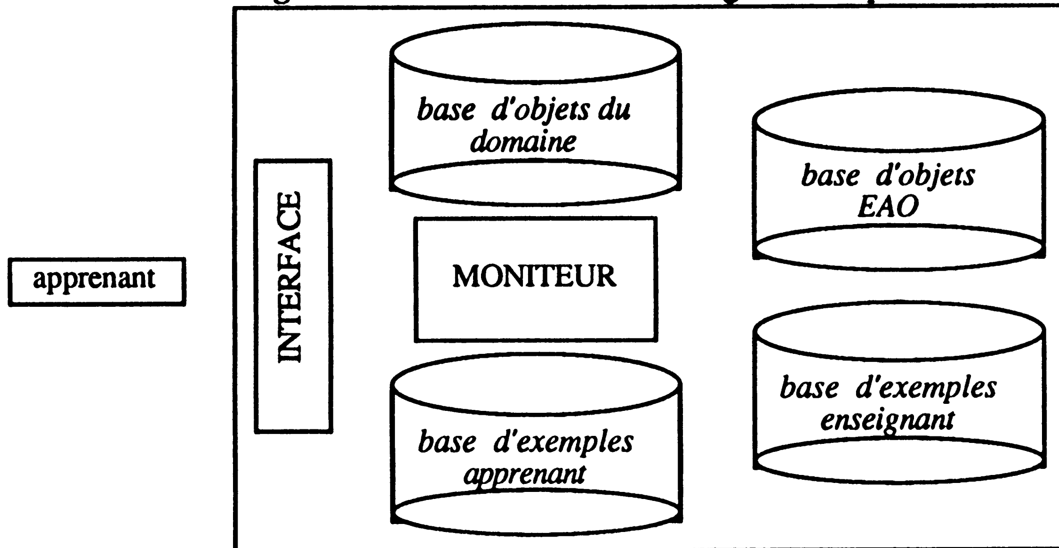
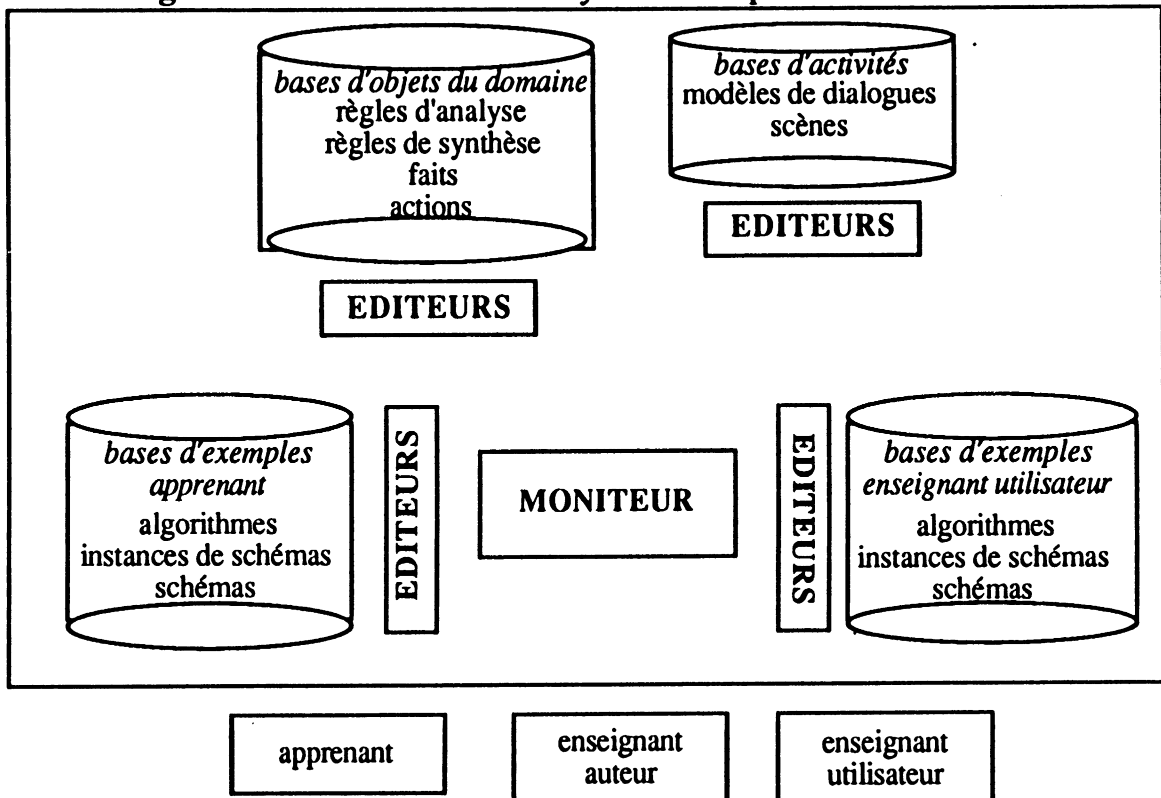


figure 4.2: architecture du système de production



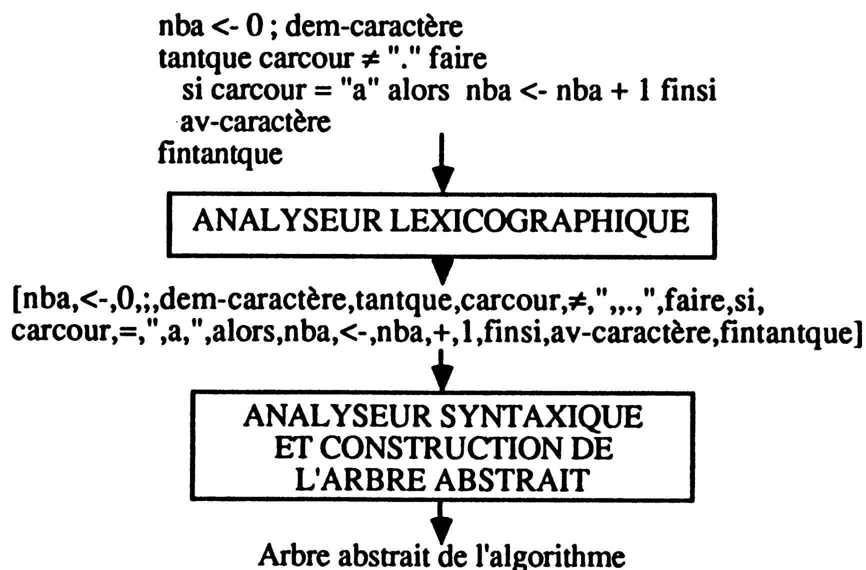
L'enseignant-auteur donne les modèles de dialogues, et les connaissances sur la notation algorithmique. L'enseignant utilisateur s'approprie la notation algorithmique et les modèles de dialogues et crée sa base d'exemples. L'apprenant exploite ces exemples et peut créer sa propre base d'exemples.

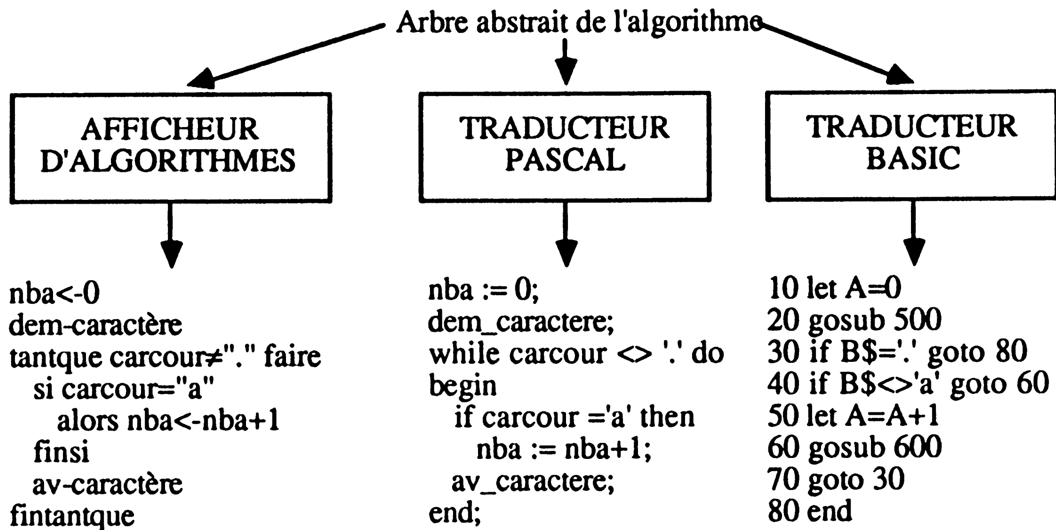
4.1.3. Expression des connaissances

L'environnement de production doit essentiellement faciliter l'expression, l'édition et l'exploitation des différentes bases de connaissances sur la notation algorithmique. Le choix de Prolog a été guidé ici par les travaux de recherche l'utilisant pour la spécification des langages de programmation. C'est ce que nous mettons en évidence, en présentant au préalable les formalismes utilisés pour description de la syntaxe et de la sémantique des langages de programmation et un exemple d'environnement de définition de langages.

L'ensemble des outils à réaliser, peut être vu comme un environnement de programmation centré sur une représentation interne: l'arbre abstrait de l'algorithme manipulé. Celui-ci est construit par un analyseur syntaxique. Les autres outils correspondent à différentes formes de décompilation ou d'interprétation de cet arbre abstrait.

figure 4.3: exemples d'outils de manipulations d'algorithmes





Les autres outils à décrire, vérificateur de types et interpréteur, peuvent être exprimés respectivement sous la forme d'une évaluation et d'une interprétation de l'arbre abstrait de l'algorithme.

Nous rappelons rapidement les méthodes de formalisation de ces différents aspects d'un langage de programmation: lexicographie (pour sa spécification de l'analyseur lexicographique), syntaxe (pour l'analyseur syntaxique), règles de présentation (pour l'afficheur), sémantique statique (pour le vérificateur de types), sémantique opérationnelle (pour l'interpréteur), règles de traduction Basic et Pascal (pour les traducteurs Basic et Pascal).

a) spécifications formelles des langages de programmation

La formalisation de la syntaxe des langages de programmation est plus simple que la formalisation de leur sémantique. Parmi les méthodes de spécification syntaxiques, la plus ancienne ([Bac60]), utilisant la notation BNF (Backus-Naur Form), est toujours largement connue et utilisée. Dans ce formalisme, la syntaxe de la construction itérative "tantque", est décrite par la règle suivante:

<instruction tantque> ::= tantque <expression> faire <algorithme> fintantque

Les méthodes de formalisation de la sémantique d'un langage sont généralement définies à partir de sa syntaxe abstraite. On distingue donc la syntaxe concrète (représentation linéaire des programmes) et la syntaxe abstraite ([LLS 68]), langage pivot dans lequel les programmes sont des arbres. Cette démarche est similaire à celle utilisée en linguistique ([Cho57]) où pour définir la sémantique d'une langue naturelle, on distingue les notions de structure de surface et de structure profonde.

syntaxe concrète: tantque expression faire algorithme fintantque
syntaxe abstraite: tantque(expression, algorithme)

Les trois principales approches pour la formalisation de la sémantique des langages de programmation sont ([Liv78], [Pag81]) l'*approche interprétative* ou opérationnelle, l'*approche dénotationnelle* et l'*approche axiomatique*. Dans notre contexte, nous ne nous intéressons pour l'instant qu'à la première. La sémantique interprétative de la construction "tantque" peut, par exemple, être donnée sous la forme suivante:

```
<tantque Expression faire Algorithme fintantque; Suite_algorithme/Environnement> —>
    soit v = Evaluation(Expression, Environnement)
    si v = vrai alors <Algorithme; tantque Expression faire Algorithme fintantque;
                Suite_algorithme/Environnement>
    si v = faux alors <Suite_algorithme/Environnement>
```

En complément de ces trois approches, la sémantique naturelle ([Kah87]), inspirée de l'approche structurale de la sémantique proposée par G.D. Plotkin dans [Plo81], permet de présenter, de manière unifiée, les différents aspects de la sémantique des langages de programmation: sémantique statique, sémantique dynamique et sémantique de traduction.

- sémantique statique

```
Env/- EXP : t, n   Env/- ALG
_____________________________
Env/- tantque EXP faire ALG fintantque :
 {if t # 'bool' then printerror ('type of expression must be boolean', EXP)}
```

- sémantique interprétative

Env/- EXP : "vrai" Env/- ALG : Env1 Env1/- tantque EXP faire ALG fintantque : Env2
____
Env/- tantque EXP faire ALG fintantque : Env2

Env/- EXP : "faux"
____
Env/- tantque EXP faire ALG fintantque : Env

- sémantique de traduction

/- EXP : c1 /- ALG : c2 c = coms[lbl1; c1; fjp 2; c2; ujp 1; lbl 2]
____
/- tantque EXP faire ALG fintantque : block(c)

b) un exemple d'environnement de définition de langages

Mentor-Typol ([CDD85], [Mel86], [Des86]) développé à l'INRIA, est un environnement de définition de langages. Il permet de générer à partir de spécifications de haut niveau du langage, un ensemble d'outils de programmation et de manipulation de programmes.

Le *sous système Mentor* permet à partir des *spécifications syntaxiques* du langage de générer analyseur et éditeur syntaxique. Pour spécifier un langage A, on doit construire un programme Metal qui comporte:
- un ensemble de règles décrivant la *syntaxe concrète* du langage A, utilisées pour la construction d'un analyseur syntaxique de A,
- un ensemble de règles décrivant la *syntaxe abstraite* du langage A,
- un ensemble de fonctions de constructions des arbres de syntaxe abstraite qui font le lien entre la forme concrète, textuelle, et la forme abstraite, arborescente, d'une expression du langage A. Ces fonctions permettent de dériver "le constructeur" de la représentation arborescente à partir des résultats produits par l'analyseur.
- un ensemble de fonctions de décompilation des arbres qui font le lien entre la forme arborescente et la forme textuelle. A partir de ces fonctions sera construit le décompilateur du langage A.

Le *sous-système Typol* permet quant à lui, de générer vérificateur de types, interpréteur, traducteurs et compilateurs, à partir des *spécifications sémantiques* du langage.

c) spécifications formelles exécutables, en Prolog, des langages de programmation.

Les premiers travaux d'A. Colmerauer et de D. Warren ([Col75], [War80]) portaient sur le traitement des langages de programmation en Prolog. Ils concernaient la réalisation d'un compilateur en montrant l'expression en Prolog des différentes phases du processus de compilation (analyse lexicale et syntaxique, construction de l'arbre abstrait et de la table des symboles, génération de code) transformant l'arbre abstrait en code machine. Ces travaux ne relatent pas une réalisation effective d'un compilateur "réel", mais montrent plutôt les avantages de l'expression Prolog pour l'écriture d'un compilateur "simple". Dans [Mon84], J.F. Monin présente les apports du langage Prolog pour l'écriture d'un compilateur "réel" dans le cadre du projet Veda.

Dans [Sim81] M. Simonet propose de définir un langage au moyen d'une W-grammaire et sa transcription en Prolog. L'objectif est d'obtenir une implantation "prototype" de langages et de valider leur définition. La *sémantique* des langages est exprimée de manière *opérationnelle*. Trois expériences de définition et d'implantation de langages sont relatées à propos de Asple, Algol 68 et Fortish.

Dans [Mos81] C.D.S Moss montre que les grammaires de métamorphose et les clauses de Horn utilisées dans le langage Prolog fournissent un formalisme puissant pour la description de différents aspects des langages de programmation, que ce soit la syntaxe, la sémantique opérationnelle ou la sémantique axiomatique.

- syntaxe

```
instruction(tantque(EXP, ALG)) --> "tantque" expression(EXP) "faire" algorithme(ALG)
                        "fintantque"
```

- sémantique opérationnelle

```
sememe(tantque(EXP, ALG)) --> sememe(EXP, val(faux)
    /sememe(EXP, val(vrai); sememe(ALG); sememe(tantque(EXP, ALG))
```

- sémantique axiomatique

La règle $\dfrac{\{P \text{ et } E\}\ A\ \{P\}}{\{P\}\ \underline{\text{tantque}}\ E\ \underline{\text{faire}}\ A\ \underline{\text{fintantque}}\ \{P \text{ et non } E\}}$

peut être traduite en Prolog sous la forme:
Axiomatique(p, tantque(EXP, ALG), p&Not(b)) <- Axiomatique(p&EXP,ALG,p)

d) formalisme choisi

Quand nous avons débuté notre travail, le système Typol n'existait pas. Nous avons développé les différentes parties du système Tangram sous forme de spécifications exécutables à l'aide de Prolog, le choix de ce langage permettant évidemment d'utiliser les acquis du système Eaolog.

Le prototype du système Tangram est actuellement disponible sur Macintosh Plus, SE,II sous l'environnement Prolog IIv2.4 de Prologia ([Pro87]).

4.2. Le système Tangram

Tangram comprend trois environnements, le premier pour l'enseignant auteur, le second pour l'enseignant utilisateur et le troisième pour l'apprenant. Plutôt que de les présenter successivement, nous décrivons les fonctionnalités qu'ils partagent, en soulignant ce qui est accessible à chacun des intervenants. Les trois principales fonctionnalités du système concernent l'expression et l'édition des connaissances sur la notation algorithmique, des modèles de dialogues des activités, des schémas, des instances et des algorithmes.

4.2.1. Connaissances sur la notation algorithmique

Lors de la phase de création l'enseignant auteur exprime les connaissances sur la notation algorithmique liées aux différents outils de

manipulations d'algorithmes que l'on souhaite fournir à l'enseignant utilisateur et à l'apprenant. Lors de la phase d'appropriation, l'enseignant utilisateur modifie ou enrichit ces connaissances (par exemple en ajoutant de nouvelles actions prédéfinies à la notation algorithmique).

Les connaissances concernant ces outils sont exprimées sous la forme d'ensembles de règles, de faits et d'actions. Nous distinguons les *règles d'analyse syntaxique et de construction de l'arbre abstrait*, et les *règles de synthèse à partir d'un arbre abstrait.*

a) règles d'analyse syntaxique et de construction de l'arbre abstrait

Nous utilisons le formalisme des grammaires de métamorphose pour l'expression de ces règles. Elles sont calquées sur les règles de la grammaire hors-contexte décrivant la syntaxe de la notation algorithmique. Pour effectuer la construction de l'arbre abstrait de l'algorithme lors de l'analyse syntaxique, il faut rajouter l'évaluation d'un attribut synthétisant la configuration d'arbre abstrait associée à la construction algorithmique analysée par la règle. Par exemple, à la construction algorithmique "tantque expression faire algorithme fintantque", nous associons la configuration d'arbre abstrait **itération1(Expression, Algorithme).**

figure 4.4: règle concernant la construction "tantque"

règle de la grammaire hors-contexte

instruction -> tantque expression faire algorithme fintantque

règle d'analyse avec évaluation de l'arbre abstrait

```
instruction (itération1(Expression,Algorithme)) -->
    mot_clé("it1")
    expression(Expression)
    mot_clé("it2")
    algorithme(Algorithme)
    mot_clé("it3")
```

Le prédicat **mot_clé** fait référence à une base de faits décrivant le vocabulaire algorithmique, associant au nom interne du mot clé ("**it1**" par exemple) les noms externes dans différentes langues (français, anglais, espagnol).

extrait du vocabulaire algorithmique

vocabulaire("it1", "tantque", "while", "mientras")
vocabulaire("it2", "faire", "do", "hacer")
vocabulaire("it3", "fintantque", "endwhile", "fmientras")

Cette séparation du vocabulaire algorithmique des règles d'analyse syntaxique permet à l'enseignant utilisateur de le modifier facilement sans accéder aux règles syntaxiques.

L'enseignant auteur utilise ce formalisme pour spécifier la syntaxe de la notation algorithmique en donnant les règles syntaxiques des déclarations, des instructions et des expressions. Pour les instructions, il y a autant de règles que de constructions algorithmiques dans la notation, et à chacune d'elles correspond une configuration d'arbre abstrait.

b) règles de synthèse à partir d'un arbre abstrait

La spécification de la décompilation (présentation d'algorithme), de la sémantique statique (vérification de types), de la sémantique dynamique (interprétation) et de la sémantique de traduction (traduction Pascal et Basic) s'exprime dans un formalisme unique. Il permet l'expression de règles associant à une configuration d'arbre abstrait, une suite de tranformations:

transformation(**configuration d'arbre abstrait**, autres attributs synthétisés ou hérités)
-> suite de transformations à effectuer

Les transformations associées à une configuration d'arbre abstrait peuvent

être des actions de visualisation comme dans le cas des règles de présentation d'un algorithme. Par exemple, la règle de présentation de la construction "tantque" s'écrit:

<u>afficher_algo</u>(**itération(Expr, Algo)**, Lig1, Col1, Lig4, Asp) :-
pca(Lig1, Col1, Aspect), aff_mot_clé("it1"),
<u>afficher_expression</u>(**Expr**, 0), aff_mot_clé("it2"),
Lig2 <- Lig1 + 1, Col2 <- Col1 + 2
<u>afficher_algo</u>(**Algo**, Lig2, Col2, Lig3, Asp)
pca(Lig3, Col1, Aspect), aff_mot_clé("it3")
Lig4 <- Lig3 + 1.

Remarques: **Lig1,Col1** et **Asp** sont des attributs hérités, ils correspondent respectivement à la ligne et la colonne de début de visualisation et à l'aspect d'affichage; **Lig4** est un attribut synthétisé correspondant à la ligne qui suit la dernière ligne affichée. **pca** est une action de positionnement du curseur et de l'aspect et **aff-mot-clé** une action d'affichage du nom externe du mot clé dans la langue courante. Cette règle n'est qu'une version simplifiée de la règle effective qui synthètise également l'arbre des zones d'affichage de la construction algorithmique comme nous le verrons dans la suite.

Pour faciliter l'expression de la transformation effectuée, dans la partie droite des règles de synthèse, le système fournit un certain nombre de fonctions, d'actions de visualisation ou de réception d'informations et d'opérateurs.

exemples d'actions, de fonctions et d'opérateurs disponibles

<u>actions</u>: pca, aff-mot-clé, inverser-zones, lire-zone, etc,
<u>fonctions</u>: conversions chaîne-entier, entier-chaîne, chaîne-réel, réel-chaîne; recherche d'un élément dans une liste; longueur d'une liste; concaténation de deux listes; etc,
<u>opérateurs</u>: +, -, *, /, et, ou, non, =, ≠, >, <, ≥, ≤, etc.

D'autre part nous souhaitons, pour faciliter la gestion des règles, n'avoir qu'une seule règle par configuration d'arbre abstrait. Nous avons donc défini une construction "si alors sinon finsi" utilisable en partie droite d'une règle de synthèse.

Par exemple, la règle d'interprétation de la construction algorithmique "tantque" s'écrit (version simplifiée):

```
interpréter_algorithme(itération(Expr,Algo), Env1, Env3) ->
     interpréter_expression(Expression,Valeur,Env1)
     si      Valeur = "faux"
     alors   Env3 <- Env1
     sinon   interpréter_algorithme(Algo, Env1, Env2)
             interpréter_algorithme(itération(Expr,Algo), Env2, Env3)
     finsi
```

Sans cette construction, nous aurions du écrire les deux règles suivantes:

```
interpréter_algorithme(itération(Expr,Algo), ..., Env1, Env1) ->
    interpréter_expression(Expression,"faux",Env1) /
interpréter_algorithme(itération(Expr,Algo), ..., Env1, Env3) ->
    interpréter_expression(Expression,"vrai",Env1)
    interpréter_algorithme(Algo, ..., Env1, Env2)
    interpréter_algorithme(itération(Expr,Algo), ..., Env2, Env3)
```

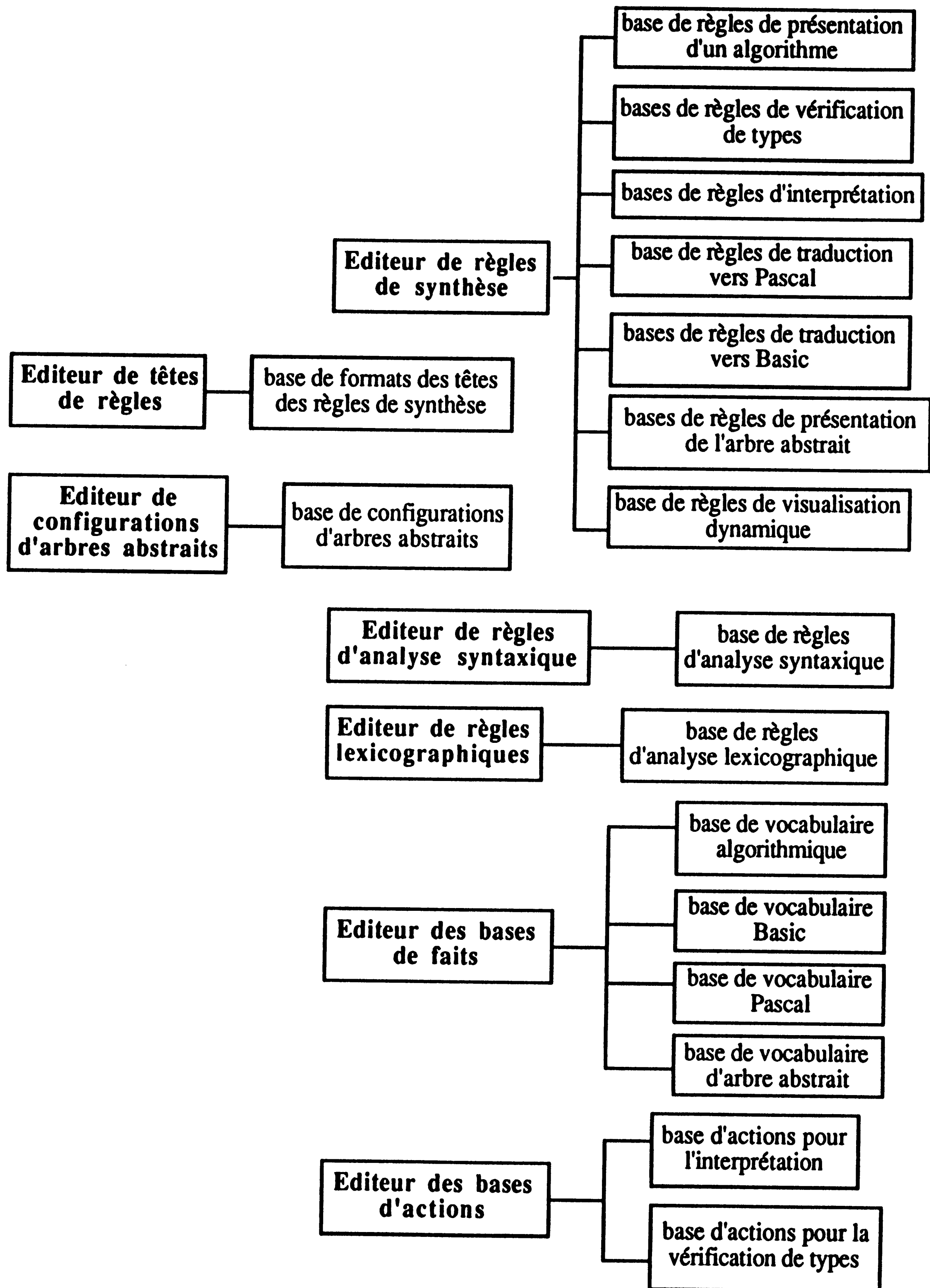
c) faits et actions

Les connaissances sur la notation exprimées sous la forme de règles d'analyse et de règles de synthèse sont éventuellement complétées par des ensembles de faits ou d'actions. C'est le cas par exemple, comme nous l'avons vu ci-dessus du vocabulaire algorithmique exprimé sous la forme d'un ensemble de faits. Cet ensemble vient également compléter les règles de synthèse exprimant la présentation d'un algorithme qui utilisent aussi les informations données dans ce vocabulaire. Les vocabulaires Basic et Pascal sont également exprimés sous cette forme. Les ensembles d'actions concernent les connaissances sur les actions prédéfinies de la notation (demcaractère, avcaractère, initdessin, gauche, droite, haut, bas, etc).

d) environnement d'édition

Les connaissances sur la notation algorithmique sont stockées dans différentes bases de règles d'analyse ou de synthèse, différentes bases de faits et d'actions. Ces bases sont modifiables au travers d'éditeurs.

figure 4.5: Environnement d'édition des connaissances sur la notation.



Nous détaillons maintenant, en donnant des exemples d'écrans d'édition, le fonctionnement de l'éditeur de règles de synthèse.

Cet éditeur demande d'indiquer la base de règles de synthèse que l'on souhaite éditer: règles de présentation, de vérification de types, d'interprétation, etc. Ayant choisi une base, on dispose de fonctions de création, suppression, modification, consultation et visualisation d'une règle. Les deux dernières fonctions sont différentes, au sens où la visualisation d'une règle correspond à visualiser l'application de cette règle sur une configuration d'arbre abstrait symbolique.

On choisit une règle en indiquant une des différentes constructions algorithmiques disponibles. Ces constructions sont réparties en trois groupes correspondant aux déclarations, aux instructions et aux expressions (sur les écrans qui suivent celles-ci sont réparties en expressions1 et expressions2).

Pour l'exemple donné dans l'écran 1, l'utilisateur a choisi de consulter la règle de traduction vers Basic de la construction algorithmique "tantque" qui appartient au groupe "instructions".

Pour l'exemple donné dans l'écran 2, l'utilisateur a choisi de visualiser l'application de cette règle. Les exemples suivants (écrans 3 et 4) correspondent respectivement à la consultation d'une règle d'interprétation, et d'une règle de vérification de types.

Ecran 1: consultation d'une règle de traduction vers Basic

Edition de la sémantique de traduction Basic

		Règle
retour	retour	bas(Lig1,Col1,iter(Exp,Alg),Lig2,
création	itérer	si
suppression	■ tantque	alors
modification	répéter	pca(Lig1,Col1,Asp)
■ consultation	si alors finsi	aba3(Lig1)
visualisation	si alors sinon finsi	vob("cd1",Var7)
retour	affectation	ecr(" ")
déclarations	lire	ecr(Var7)
■ instructions	écrire	ecr(" ")
expressions1	liste instructions	obas("non",Var8,Var9)
expressions2	liste instruction	ecr(Var9)
		ecr("(")
		bas(Exp,0,Lvb)
		ecr(")")
		vob("allera",Var10)
		ecr(" ")

Ecran 2: visualisation de l'application d'une règle de traduction vers Basic

Edition de la sémantique de traduction Basic

		Règle
retour	retour	
création	itérer	tantque expression faire
suppression	■ tantque	actions
modification	répéter	ftantque
consultation	si alors finsi	
■ visualisation	si alors sinon finsi	100 IF not(expression') GOTO 130
retour	affectation	110 GOSUB actions'
déclarations	lire	120 GOTO 100
■ instructions	écrire	
expressions1	liste instructions	
expressions2	liste instruction	

Ecran 3: consultation d'une règle d'interprétation

Edition de la sémantique interprétative

retour	retour	Règle
création	itérer	
suppression	tantque	
modification	répéter	
■ consultation	si alors finsi	
visualisation	■ si alors sinon finsi	
retour	affectation	
déclarations	lire	
■ instructions	écrire	
expressions1	liste instructions	
expressions2	liste instruction	

```
int(cond(Exp,Alg1,Alg2),cond([Var
  si
    int(Exp,Var4,Env1)
    Var4 = "vrai"
  alors
    ua9(3,Asp,[Var1])
    ua9(3,Asp,[Var1])
    int(Alg1,Var2,Asp,Env1,Env2)
  sinon
    ua9(3,Asp,[Var1])
    ua9(3,Asp,[Var1])
    int(Alg2,Var3,Asp,Env1,Env2)
  fsi
```

Ecran 4: consultation d'une règle de vérification de types

retour	retour	Règle
création	constante	
suppression	variable	
modification	expr binaire +	
■ consultation	expr binaire -	
visualisation	expr binaire *	
retour	expr binaire /	
déclarations	expr unaire -	
instructions		
■ expressions1		
expressions2		

```
typ(exp("*",Exp1,Exp2),Type,Env):
  si
    typ(Exp1,"ent",Env)
    typ(Exp2,"ent",Env)
  alors
    Type<-"ent"
  sinon
    Type<-"err"
  fsi
```

e) indications sur la réalisation des éditeurs

Considérons par exemple l'édition de l'ensemble des bases de règles de synthèse. Nous avons réalisé un seul éditeur qui utilise la base de configurations d'arbres abstraits (constructions algorithmiques disponibles) et la base de formats des règles de synthèse. A chaque configuration d'arbre abstrait correspond une règle unique dans chacune des bases servant à la synthèse (cf figure 4.5). Pour chacune d'elles, nous distinguons trois formats de règles de synthèse pour les déclarations, les instructions et les expressions.

L'éditeur assure une fonction de gestion (ajout, suppression, modification) des bases de règles de synthèse et une fonction de traduction: passage de la forme externe d'une règle de synthèse à sa forme interne. Pour assurer cette fonction de traduction l'éditeur est lui-même composé d'une base de règles d'analyse (analyse syntaxique d'une règle de synthèse) et d'une base de règles de synthèse (présentation d'une règle de synthèse).

A titre d'exemple, nous donnons deux exemples de forme externe de règles de synthèse et des formes internes produites par l'éditeur. Le premier exemple concerne l'évaluation des types d'une expression additive:

forme externe
```
eval_types(expression("+",Exp1,Exp2),Type,Env):
    si eval_types(Exp1,"entier",Env) eval_types(Exp2,"entier",Env)
        alors Type <- "entier"
        sinon Type <- "erreur de type"
    finsi
```
forme interne:
```
eval_types(expression("+",X,Y),T,E) ->
    si(eval_types(X,"ent",E).eval_types(Y,"ent",E).nil,
        ut("<-",T,"ent").nil,    ut("<-",T,"err").nil) /;
```
format de la tête de la règle de synthèse utilisée:
```
str-bco(A-rb,"1","3",eval-types(A-rb,X2,X3),eval-types(A-rb,T,E)) ->;
```
format de la configuration d'arbre abstrait utilisée:
```
str-caa("3","expr binaire +",expression( "+",X2,X3),expression("+",Exp1,Exp2)) ->;
```

Le deuxième exemple, un peu plus complexe, concerne l'interprétation de l'instruction conditionnelle:

<u>forme externe</u>

```
interprétation(cond(Expr,Algo1,Algo2),cond([Var1],Var2,Var3),Asp,Env1,Env2):
          si  interprétation (Expr,Var4,Env1)
              Var4 = "vrai"
          alors
               inverser_liste_zones(3,Asp,[Var1])
               inverser_liste_zones (3,Asp,[Var1])
               interprétation (Algo1,Var2,Asp,Env1,Env2)
          sinon
               inverser_liste_zones (3,Asp,[Var1])
               inverser_liste_zones (3,Asp,[Var1])
               interprétation (Algo2,Var3,Asp,Env1,Env2)
           fsi
```

<u>forme interne</u>

```
int(cond(e1,i1,i2),cond(z-o.l-zo,z-o1,z-o2),a,l-a1,l-a2) ->
     si(int(e1,v-e1,l-a1).it("=",v-e1,"vrai").nil,ua9(3,a,z-o.nil).ua9(3,a,z-o.nil).int(i1,
     z-o1,a,l-a1,l-a2).nil,ua9(3,a,z-o.nil).ua9(3,a,z-o.nil).int(i2,z-o2,a,l-a1,l-a2).nil) /;
```

4.2.2. Modèles de dialogues des activités

L'enseignant-auteur fournit au système les modèles de dialogues des activités qu'il propose à l'apprenant.

a) moyen d'expression:

L'enseignant-auteur dispose du langage auteur du système Eaolog, complété par des actions de manipulations d'algorithmes. En effet les ensembles de règles du paragraphe précédent correspondent à une extension des *actions disponibles* pour l'expression des dialogues. Ces actions sont applicables sur des objets de type arbre abstrait. Nous illustrons ceci avec le modèle de dialogue associé à l'activité de traduction d'un

algorithme en Basic. Rappelons que le dialogue se compose d'un écran de présentation de l'activité, puis d'une traduction pas à pas de chaque construction de l'algorithme (cf §3.2.3 c où sont définis les écrans de ce modèle). Nous commentons dans la figure 4.6 les étapes de ce dialogue avec les conventions suivantes: les parties en italiques sont des *commentaires*, les actions de manipulation d'arbre abstrait sont soulignées, les **arbres abstraits manipulés** sont indiqués en caractères gras).

figure 4.6: visualisation d'une traduction, construction du dialogue

{CHOIX D'UN ALGORITHME DE LA BASE D'ALGORITHMES}

menu_dynamique(arb_algo, Nom_choisi)

> *{menu dynamique sur la base d'arbres abstraits d'algorithmes courante, on obtient dans la variable Nom_choisi le nom de l'arbre abstrait qui a été choisi}*

arb_algorithmes(Nom_choisi, **Arb_decl**, **Arb_alg**)

> *{on récupère dans la base d'arbres abstraits d'algorithmes, l'arbre abstrait des déclarations: Arb_decl, et l'arbre abstrait de la partie algorithme: Arb_alg}*

{AFFICHAGE D'UNE SCENE DE PRESENTATION DE L'ACTIVITE}

avancer_scène(scène_trad_basic)

> *{on affiche la scène de présentation de la traduction vers Basic}*

aspect_algorithme(Asp_alg)

> *{on récupère dans la base d'aspects, l'aspect de visualisation de l'algorithme: Asp_alg}*

point_d'affichage(pt_alg, pt(Lig1, Col1))

> *{on récupère dans la base de points d'affichage, les coordonnées ligne colonne: Lig1, Col1 du point d'affichage de l'algorithme}*

aspect_basic(Asp_basic)

> *{on récupère dans la base d'aspects, l'aspect de visualisation d'un programme Basic: Asp_basic}*

point_d'affichage(pt_basic, pt(Lig4, Col4))

> *{on récupère dans la base de points d'affichage, les coordonnées ligne colonne: Lig4, Col4 du point d'affichage du programme Basic}*

{AFFICHAGE DU TEXTE DE L'ALGORITHME}

afficher_déclarations(**Arb_decl, Azd_algo**, Asp_alg, Lig1, Col1, Lig2)

{on visualise les déclarations dont on obtient l'arbre des zones d'affichage: Azd_algo, on obtient aussi la valeur de la ligne qui suit la dernière ligne écrite: Lig2}

afficher_algorithme(**Arb_alg, Aza_algo**, Asp_alg, Lig2, Col1, Lig3)

{on visualise l'algorithme dont on obtient l'arbre des zones d'affichage: Aza_algo, on obtient la valeur de la ligne qui suit la dernière ligne écrite: Lig3}

{AFFICHAGE DU TEXTE DU PROGRAMME BASIC}

afficher_décl_basic(**Arb_decl, Azd_basic**, Asp_basic, Lig4,Col4, Lig5, [], Liste_assoc)

{on visualise la traduction Basic des déclarations dont on obtient l'arbre des zones d'affichage: Azd_basic, on obtient aussi la valeur de la ligne qui suit la dernière ligne écrite: Lig5, et la liste d'associations: Liste_assoc liant les identificateurs de l'algorithme à des noms de variables Basic}

afficher_alg_basic(**Arb_alg, Aza_basic**, Asp_basic, Lig5, Col4, Lig6, Liste_assoc)

{on visualise la traduction Basic de l'algorithme dont on obtient l'arbre des zones d'affichage: Aza_basic, on obtient aussi la valeur de la ligne qui suit la dernière ligne écrite: Lig6}

{VISUALISATION DE LA MISE EN CORRESPONDANCE ENTRE l'ALGORITHME ET LE PROGRAMME BASIC}

afficher_correspondance(**Azd_algo, Azd_basic**, Asp_alg, Asp_basic)

{on visualise la mise en correspondance entre la partie déclaration de l'algorithme et la traduction Basic associée}

afficher_correspondance(**Aza_algo, Aza_basic**, Asp_alg, Asp_basic)

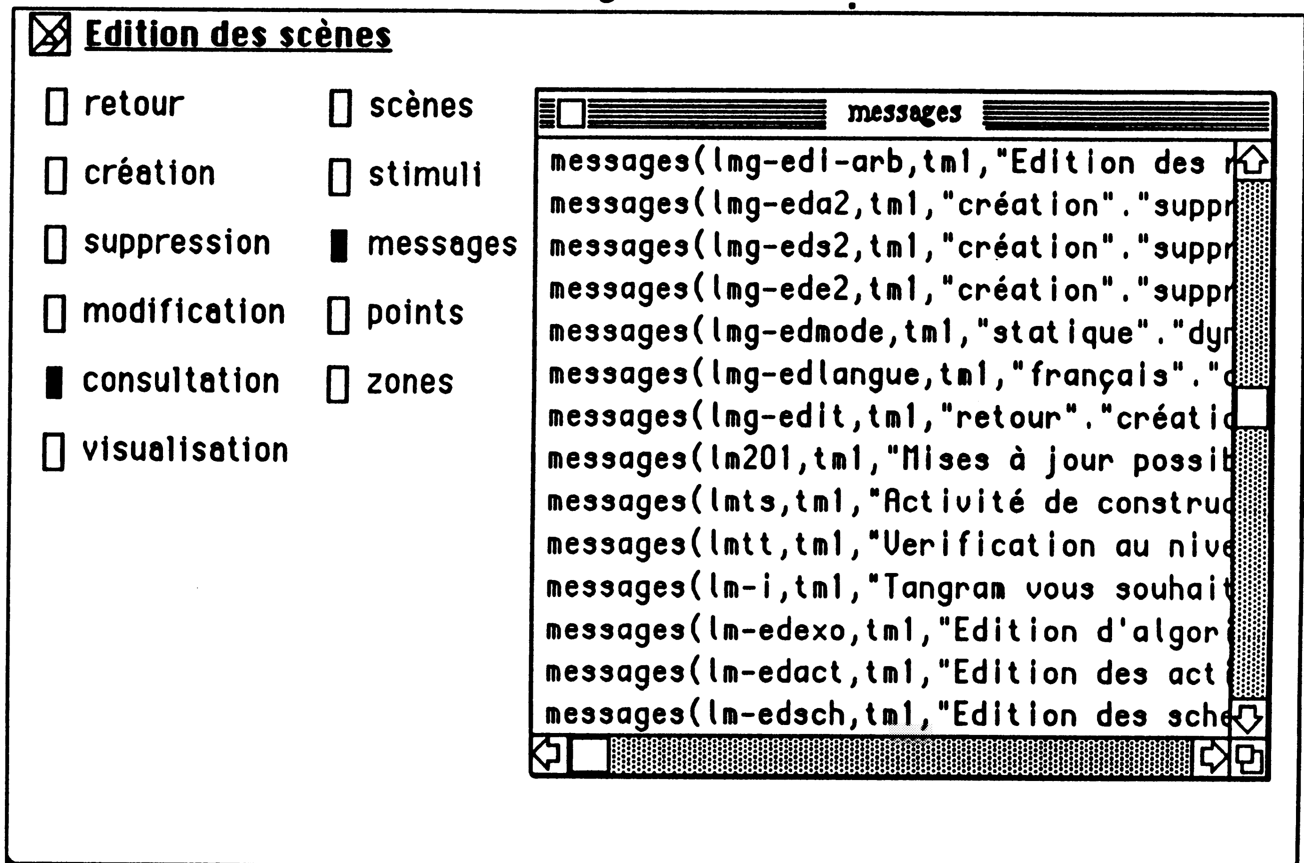
{on visualise la mise en correspondance entre la partie algorithme et la traduction Basic associée}

b) environnement d'édition

Les outils d'édition correspondent à l'éditeur de dialogues et éditeur d'objets E.A.O. présentés au chapitre II. Par exemple, l'écran 1 ci-dessous est extrait d'une session d'édition des scènes d'un dialogue. On voit que l'utilisateur dispose d'un premier menu de fonctions: création, suppression, etc. Il a choisi ici la fonction de consultation et il indique ensuite le type

des objets E.A.O. qu'il souhaite consulter: scènes, stimuli, etc. Il a choisi ici de consulter les messages, qu'il obtient dans la fenêtre "messages".

Ecran 1: Edition des messages des scènes



En plus de cet éditeur d'objets E.A.O, l'enseignant dispose d'un éditeur d'aspects d'affichage. Les écrans 2 et 3 montrent justement un écran apprenant, avant et après une modification d'aspects de l'algorithme et du programme.

Ecran 2: écran apprenant avant la modification d'aspects

Activité de traduction en Basic d'un algorithme

```
nommer logique trouve;
nommer caractere carcour;
trouve<-faux
demcaractere
tantque (carcour ≠ '.') et non trouve faire
   si carcour = 'a'
      alors   trouve<-vrai
      sinon   avcaractere
   fsi
ftantque
si trouve
   alors   ecrire(vrai)
   sinon   ecrire(faux)
fsi
```

```
30 REM boolean t1
40 REM char c2$
50 LET t1=false
60 GOSUB demcaractere
70 IF not((c2$ ≠ '.') and not t1) GOTO 130
80 IF not(c2$ = 'a') GOTO 110
90 LET t1=true
100 GOTO 120
110 GOSUB avcaractere
120 GOTO 70
130 IF not(t1) GOTO 160
140 PRINT true
150 GOTO 170
160 PRINT false
170 END
```

-->

Ecran 3: écran apprenant aprés la modification d'aspects

Activité de traduction en Basic d'un algorithme

```
nommer logique trouve;
nommer caractere carcour;
trouve<-faux
demcaractere
tantque (carcour ≠ '.')etnontrouvefaire
   si carcour = 'a'
      alors   trouve<-vrai
      sinon   avcaractere
   fsi
ftantque
si trouve
   alors   ecrire(vrai)
   sinon   ecrire(faux)
fsi
```

```
30 REM boolean t1
40 REM char c2$
50 LET t1=false
60 GOSUB demcaractere
70 IF not((c2$ ≠ '.') and not t1)GOTO130
80 IF not(c2$ = 'a') GOTO 110
90 LET t1=true
100 GOTO 120
110 GOSUB avcaractere
120 GOTO 70
130 IF not(t1) GOTO 160
140 PRINT true
150 GOTO 170
160 PRINT false
170 END
```

-->

4.2.3. Schémas, instances et algorithmes

L'enseignant utilisateur fournit au système des schémas, des instances de schémas et des algorithmes qui seront disponibles pour l'apprenant au travers des différentes activités spécifiées par l'enseignant auteur. L'apprenant peut de même construire ses propres exemples.

Pour exprimer les schémas, les instances et les algorithmes, l'enseignant utilisateur ou l'apprenant utilisent *la notation algorithmique* . Les exemples de schémas, d'instances et d'algorithmes, donnés par l'enseignant-utilisateur ou par l'apprenant, sont stockées dans trois bases différentes, modifiables au travers de trois éditeurs: éditeur de schémas, éditeur d'instances, éditeur d'algorithmes. Ces trois éditeurs utilisent l'analyseur syntaxique, le vérificateur de types et l'afficheur de l'environnement de manipulation d'algorithmes. Nous donnons des exemples d'écrans d'édition, pour chacun de ces trois éditeurs.

a) édition des schémas

L'utilisateur donne le nom du schéma, les noms des parties concernant l'accés et le traitement, puis l'algorithme du schéma.

Ecran 1

Edition des schémas d'algorithmes

création
suppression
modification
consultation
retour

nom du schéma
NOM
noms des parties (accés à la file)
NOMS DES PARTIES
noms des parties (traitement)
NOMS DES PARTIES
algorithme du schéma
ALGORITHME
DU SCHEMA

-->

Ecran 2

Edition des schémas d'algorithmes

■ création	**nom du schéma**	**schéma présenté**
☐ suppression	NOM	ALGORITHME DU SCHEMA PRESENTE
☐ modification	**noms des parties (accés à la file)**	
☐ consultation	NOMS DES PARTIES	
☐ retour	**noms des parties (traitement)**	
	NOMS DES PARTIES	
	algorithme du schéma	
	ALGORITHME DU SCHEMA	

--> ☐

Un exemple

Ecran 1:

Edition des schémas d'algorithmes

■ création	**nom du schéma**
☐ suppression	parcours de file
☐ modification	**noms des parties (accés à la file)**
☐ consultation	"dem" "av" "non fdf"
☐ retour	**noms des parties (traitement)**
	"initp" "termp" "visiter"
	algorithme du schéma

```
initp dem
tantque non fdf faire
   visiter av
ftantque
termp
```

--> ☐

Ecran 2

Edition des schémas d'algorithmes

- création
- suppression
- modification
- consultation
- retour

nom du schéma

parcours de file

noms des parties (accés à la file)

"dem" "av" "non fdf"

noms des parties (traitement)

"initp" "termp" "visiter"

algorithme du schéma

```
initp dem
tantque non fdf faire
   visiter av
ftantque
termp
```

schéma présenté

```
declarations
initp
dem
tantque non fdf faire
   visiter
   av
ftantque
termp
```

-->

b) édition des instances de schémas

Ecran 1:

Edition d'algorithmes instances de schémas

- création
- suppression
- modification
- consultation
- retour

Nom de l'algorithme

NOM CHOISI

Choix d'un type d'accés

TYPE DE MACHINE CHOISI

Choix d'un schéma

SCHEMA CHOISI DANS LA BASE
COURANTE DE SCHEMAS

-->

Ecran 2

Edition d'algorithmes instances de schémas

▮ création

▯ suppression

▯ modification

▯ consultation

▯ retour

Nom de l'algorithme

NOM CHOISI

Choix d'un type d'accés

TYPE DE MACHINE CHOISI

Choix d'un schéma

SCHEMA CHOISI DANS LA BASE COURANTE DE SCHEMAS

schéma choisi

ALGORITHME DU SCHEMA

--> ▯

Ecran 3

Edition d'algorithmes instances de schémas

▮ création

▯ suppression

▯ modification

▯ consultation

▯ retour

ALGORITHME DE LA 1ère PARTIE DU SCHEMA

ALGORITHME DE LA 2ème PARTIE DU SCHEMA

ETC ...

schéma utilisé

ALGORITHME DU SCHEMA

--> ▯

Ecran 4

Edition d'algorithmes instances de schémas

schéma utilisé

- création
- suppression
- modification
- consultation
- retour

algorithme obtenu

ALGORITHME OBTENU

PAR INSTANTIATION

DU SCHEMA

ALGORITHME

DU SCHEMA

-->

Un exemple

Ecran 1

Edition d'algorithmes instances de schémas

- création
- suppression
- modification
- consultation
- retour

Nom de l'algorithme

compter les 'le'

Choix d'un type d'accés

- machine de type 1
- machine de type 2

Choix d'un schéma

- parcours de file
- recherche dans une file
- parcours de sous-file définie par inclusion
- parcours de sous file définie par exclusion

-->

Ecran 2

Edition d'algorithmes instances de schémas

■ création
□ suppression
□ modification
□ consultation
□ retour

Nom de l'algorithme

compter les 'le'

Choix d'un type d'accés

■ machine de type 1
□ machine de type 2

Choix d'un schéma

■ parcours de file
□ recherche dans une file
□ parcours de sous-file définie par inclusion
□ parcours de sous file définie par exclusion

schéma utilisé

```
nommer logique fdf
initp
dem
tantque non fdf faire
   visiter
   av
ftantque
termp
```

--> □

Ecran 3

Edition d'algorithmes instances de schémas

■ création
□ suppression
□ modification
□ consultation
□ retour

déclarations

```
nommer caractere carcour, cp
nommer entier cpt
```

dem

```
cp <- '_'
demcaractere
```

av

```
cp <- carcour
avcaractere
```

non fdf

```
carcour ≠ '.'
```

initp

```
cpt <- 0
```

termp

```
ecrire(cpt)
```

visiter

```
si (cp = 'l') et (carcour = 'e')
alors cpt <- cpt + 1 fsi
```

schéma utilisé

```
nommer logique fdf
initp
dem
tantque non fdf faire
   visiter
   av
ftantque
termp
```

--> □

Ecran 4

Edition d'algorithmes instances de schémas

- création
- suppression
- modification
- consultation
- retour

algorithme obtenu

```
nommer caractere carcour,cp
nommer entier cpt
cpt <- 0
cp <- '_'
demcaractere
tantque carcour ≠ '.' faire
   si (cp = 'l') et (carcour = 'e')
      alors cpt <- cpt + 1
   fsi
   cp <- carcour
   avcaractere
ftantque
ecrire(cpt)
```

schéma utilisé

```
nommer logique fdf
initp
dem
tantque non fdf faire
   visiter
   av
ftantque
termp
```

-->

c) édition des algorithmes

En mode création, l'utilisateur donne le nom et le texte d'un l'algorithme

Ecran 1

Edition d'algorithmes directs

- création
- suppression
- modification
- consultation
- retour

nom de l'algorithme

NOM

votre algorithme

ALGORITHME

-->

Il obtient le texte présenté de cet algorithme.

Ecran 2

Edition d'algorithmes directs

- création
- suppression
- modification
- consultation
- retour

nom de l'algorithme

NOM

votre algorithme

ALGORITHME

algorithme présenté

ALGORITHME PRESENTE

-->

Un exemple sans erreurs de types

Ecran 1.1.

Edition d'algorithmes directs

- création
- suppression
- modification
- consultation
- retour

nom de l'algorithme

essai

votre algorithme

```
nommer entier i,j lire(i)
si i ≥ 0 alors j<- i sinon
j <- -i fsi
ecrire(j)
```

-->

Ecran 1.2.

Edition d'algorithmes directs

- ■ création
- □ suppression
- □ modification
- □ consultation
- □ retour

nom de l'algorithme

essai

votre algorithme

nommer entier i,j lire(i)
si i ≥ 0 alors j<- i sinon
j <- -i fsi
ecrire(j)

algorithme présenté

nommer entier i,j
lire(i)
si i ≥ 0
  alors j <- i
  sinon j <- - i
fsi
ecrire(j)

--> □

Un exemple avec des erreurs de types:

Ecran 3

Vérifications au niveau des types

ALGORITHME

EXPRESSION ERRONEE

EXPRESSION ERRONEE

ETC ...

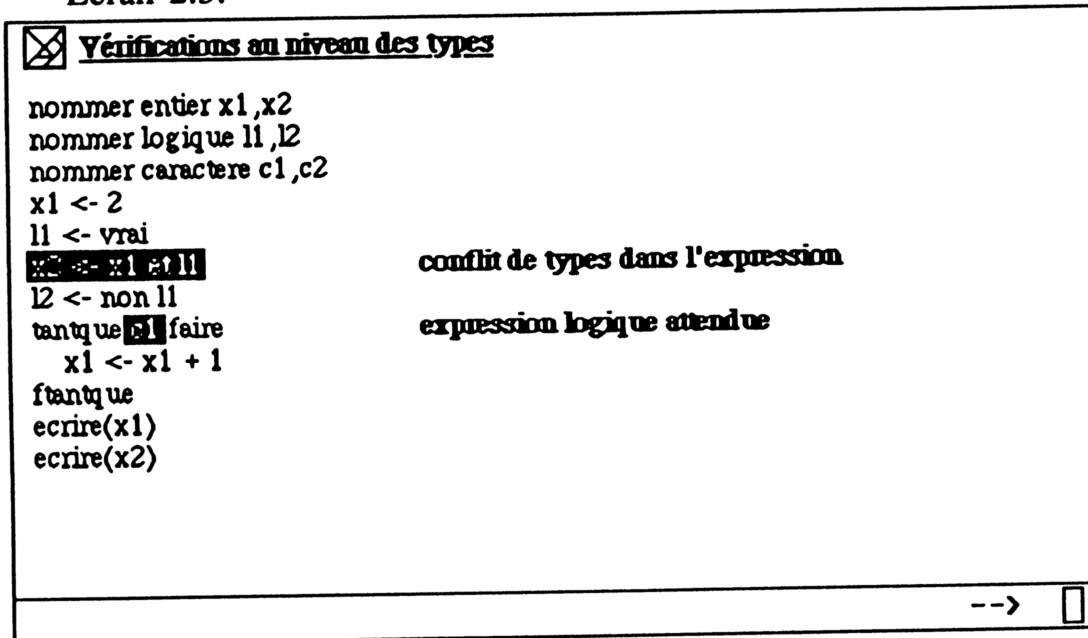
MESSAGE D'ERREUR

MESSAGE D'ERREUR

ETC ...

--> □

Ecran 2.3.



4.3. Le logiciel Idalgo

4.3.1. Fonctionnalités de l'environnement apprenant

Idalgo fournit à l'apprenant un environnement de programmation dont le langage pivot est la notation algorithmique. Il a ainsi accès à un ensemble d'outils et d'informations:

- outils: analyseur syntaxique, vérificateur de types, afficheur d'algorithme, interpréteur d'algorithmes, traducteurs Pascal et Basic
- informations: règles syntaxiques, de vérification de types, de présentation, d'interprétation, de traduction Pascal et Basic.

L'apprenant peut éditer des bases d'exemples à l'aide des éditeurs de schémas, d'instances de schémas et d'algorithmes (avec afficheur et vérificateur de types intégrés). Pour l'édition des instances de schémas, il peut travailler soit les schémas donnés par l'enseignant, soit avec ses propres schémas. Il dispose pour cela d'un ensemble d'outils permettant d'observer la construction d'une instance de schéma, d'exécuter un algorithme sur un jeu d'essai qu'il choisit, de traduire un algorithme en Pascal ou en Basic. Nous avons déja détaillé ces activités §3.2.3.

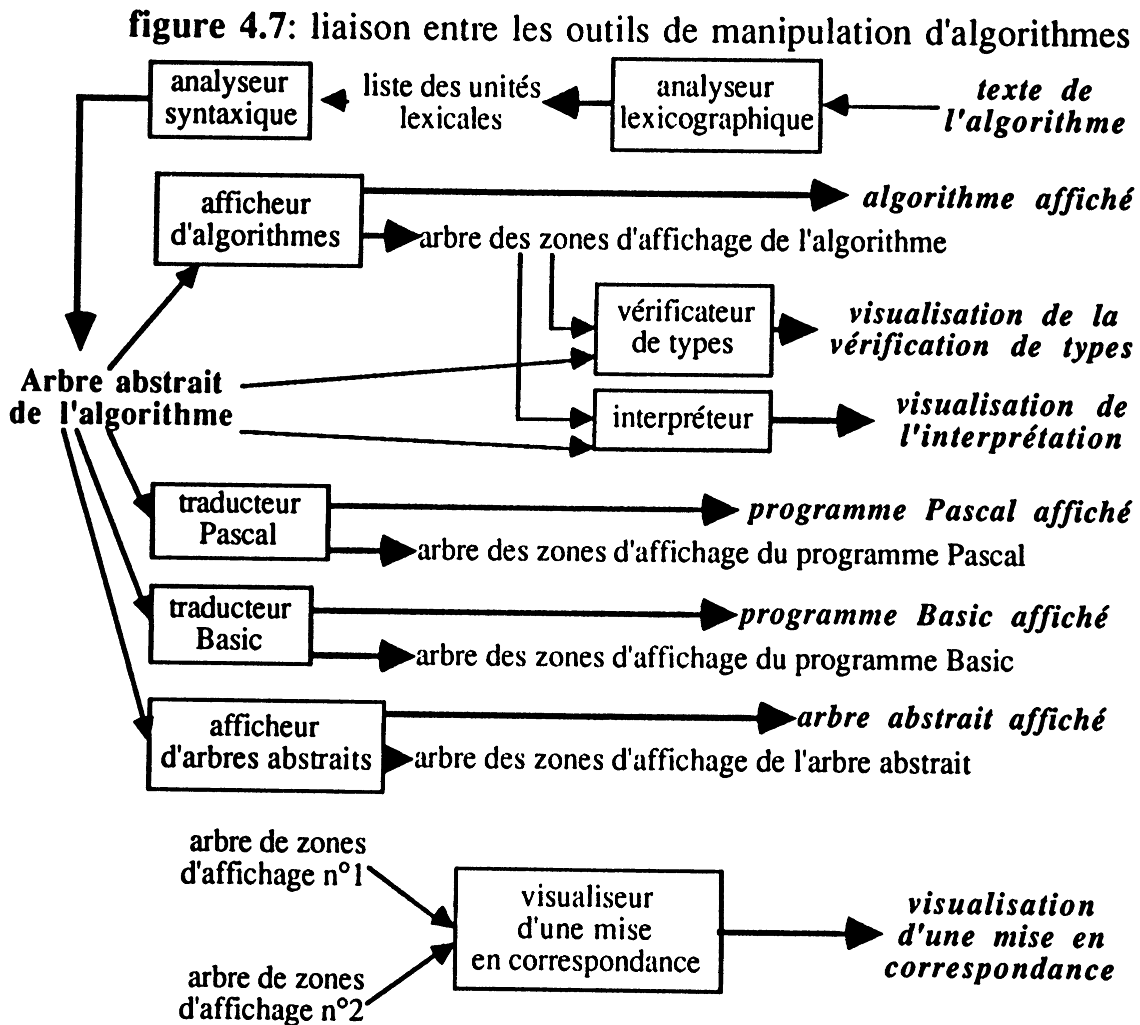
L'apprenant dispose également d'un ensemble d'informations sur la

notation algorithmique: il peut consulter la liste des constructions algorithmiques autorisées, et pour chaque construction il a accès à sa syntaxe, sa sémantique et ses traductions en Basic et en Pascal. Il dispose également de la liste des actions prédéfinies de la notation algorithmique.

4.3.2 Description des outils de manipulation d'algorithmes

La figure 4.7 illustre le rôle central de l'arbre abstrait dans la production des diverses informations visualisées pour l'apprenant.

figure 4.7: liaison entre les outils de manipulation d'algorithmes



Nous décrivons brièvement chacun des outils en faisant apparaître son mode de description en termes des bases de règles et de faits.

analyseur lexicographique

Il transforme le texte de l'algorithme en la liste de ses unités lexicales. Il est décrit par des règles d'analyse lexicographique.

analyseur syntaxique

Il transforme la liste des unités lexicales de l'algorithme en l'arbre abstrait correspondant. Il est décrit par des règles d'analyse syntaxique et de construction de l'arbre abstrait complétées par des faits décrivant les vocabulaires de mots clés (Français, Anglais, Espagnol).

afficheur

Il affiche le texte de l'algorithme à partir de son arbre abstrait. De plus il construit l'arbre des zones d'affichages de l'algorithme. Il est décrit par des règles de présentation d'algorithmes complétées par des faits décrivant les vocabulaires de mots clés.

vérificateur de types

Les erreurs détectées sont visualisées sur l'algorithme déjà affiché à l'écran en utilisant l'arbre des zones d'affichage de l'algorithme. Il est décrit par des règles de vérification de types complétées par des actions de visualisation de la vérification.

interpréteur

La visualisation de l'exécution est effectuée sur l'algorithme déja affiché à l'écran en utilisant l'arbre des zones d'affichage. Il est décrit par des règles d'interprétation, complétées par des actions de visualisation de l'interprétation.

traducteurs Pascal et Basic

Ils affichent le texte du programme Pascal ou Basic correspondant à l'arbre abstrait de l'algorithme donné en entrée, et construisent l'arbre des

zones d'affichages du programme. Il sont décrits par des règles de traduction vers Pascal ou Basic, complétées par des faits décrivant le vocabulaire Pascal ou Basic.

afficheur d'arbre abstrait

Il affiche une forme indentée de l'arbre abstrait d'un algorithme, et construit l'arbre des zones d'affichage associé. Il est décrit par des règles de présentation de l'arbre abstrait complétées par des faits décrivant le vocabulaire d'arbre abstrait.

visualiseur d'une mise en correspondance

Il visualise une mise en correspondance entre deux objets affichés séparément, au travers des arbres de zones d'affichage de ces deux objets. Cet outil peut être utilisé pour la visualisation d'une mise en correspondance entre:
- un algorithme et le programme Pascal correspondant,
- un algorithme et le programme Basic correspondant,
- un algorithme et l'arbre abstrait correspondant,
- un programme Pascal et le programme Basic correspondant, etc ...

4.3.3. Critiques et extensions d'Idalgo

La notation algorithmique proposée à l'apprenant est assez "pauvre" dans l'état actuel: elle ne permet pas la définition de types, elle n'autorise pas la définition de procédures paramétrées. Le jeu d'actions prédéfinies est limité aux actions d'accès séquentiel à des suites de caractères et d'entiers et d'une machine de tracés de dessins. L'"enrichissement" de la notation algorithmique est prévu: il suffit de compléter la base d'actions d'interprétation qui décrit la sémantique des actions prédéfinies; le passage à une notation autorisant la définition de procédures paramétrées demande plus de travail.

La base de schémas ne comporte actuellement que les schémas du traitement séquentiel, mais il serait facile d'ajouter les schémas du traitement arborescent et de compléter également la base d'instances données par l'enseignant par des exemples illustrant le traitement arborescent.

Extensions de l'environnement apprenant

En "amont" de l'expression algorithmique, des maquettes ont déja été réalisées indépendamment de notre système sur d'autres thèmes: enseignement de la récursivité et des transformations de programmes ([Gir86]), enseignement de la preuve ([Agu85]). Notre système de production permet d'intégrer ces travaux dans un environnement unique. Nous donnons des exemples extraits de ([Gir86]): l'apprenant exprime la fonction fibonacci sous la forme d'équations récurrentes (figure 4.8), il peut ensuite demander la visualisation de l'arbre des appels pour un appel initial donné (figure 4.9).

figure 4.8: édition de la fonction fibonacci

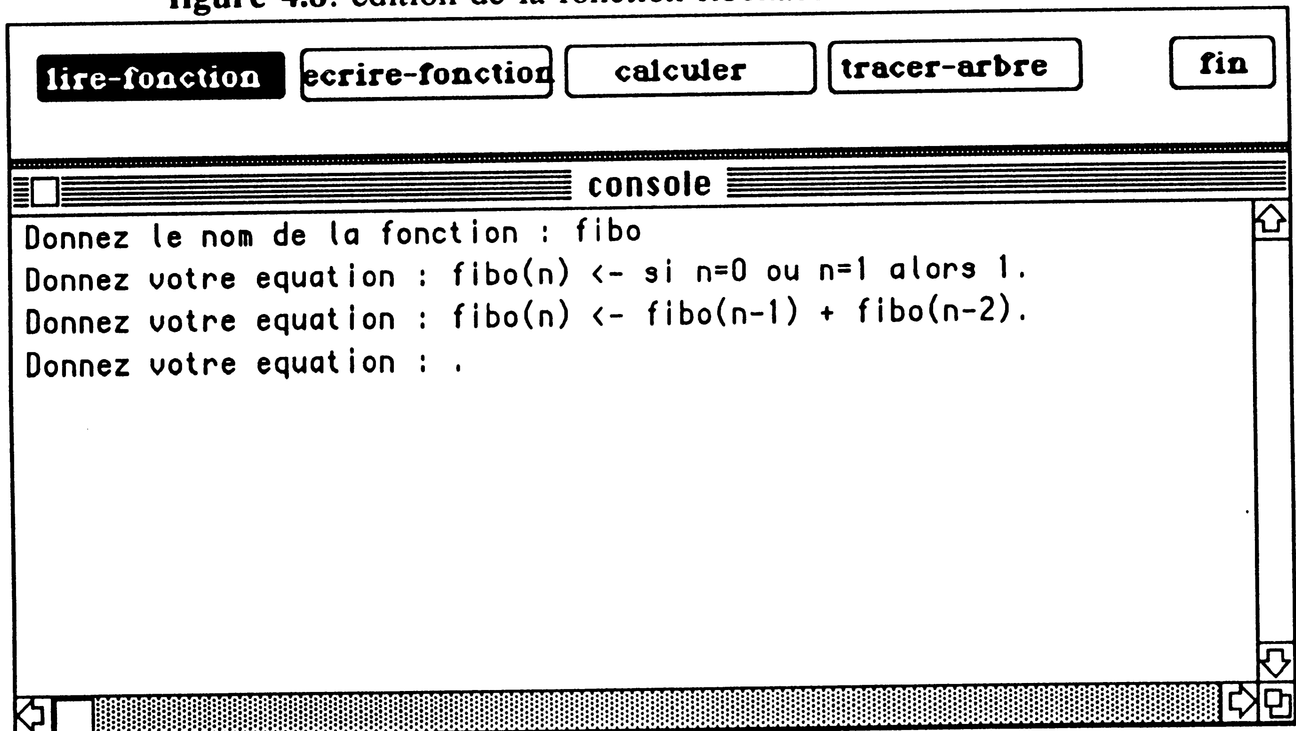
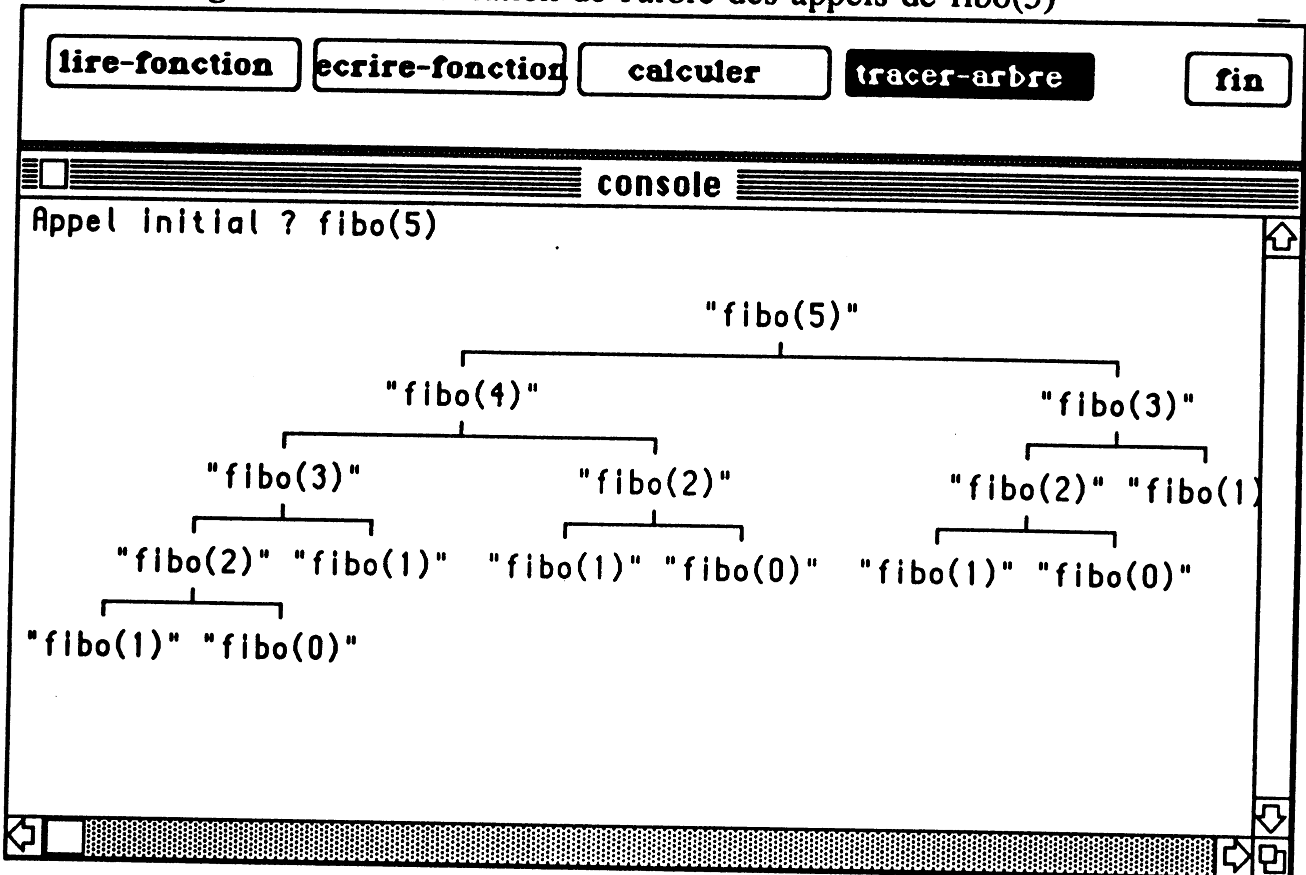


figure 4.9: visualisation de l'arbre des appels de fibo(5)

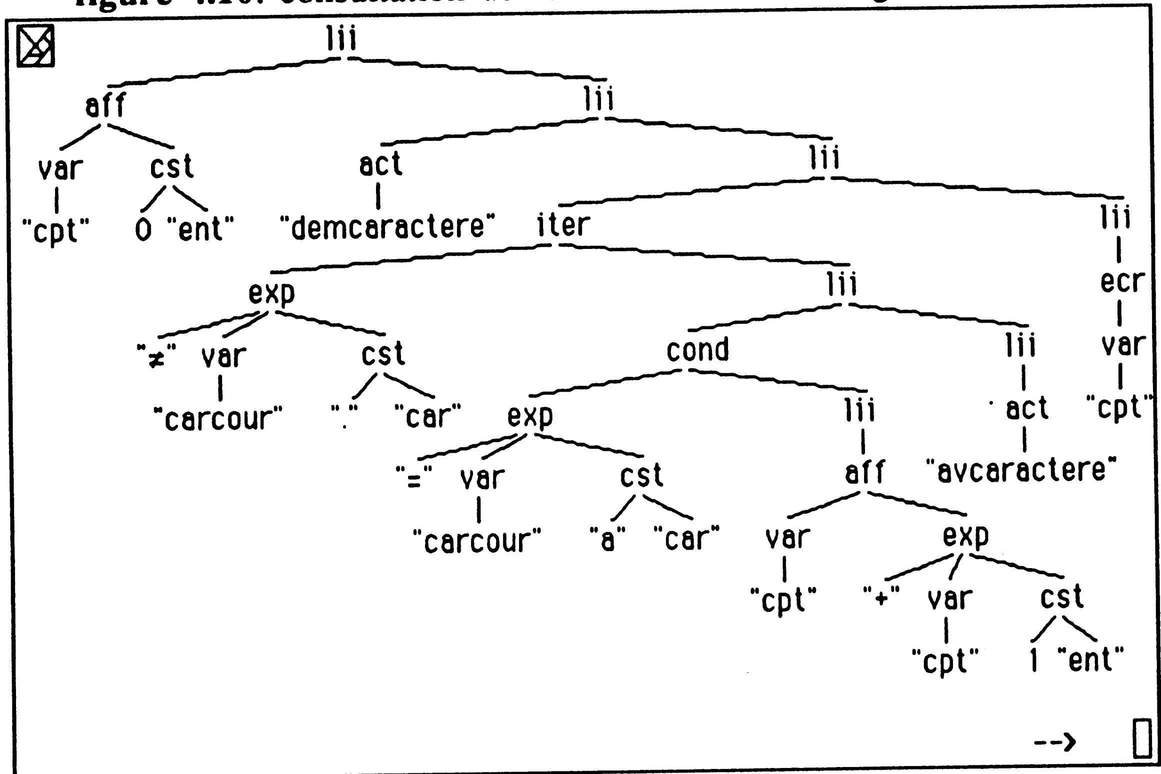


En complément de l'expression algorithmique, on pourrait définir une activité de preuve d'algorithmes, avec expression d'assertions et d'invariants. Il "suffirait" pour cela d'exprimer en Prolog la sémantique axiomatique de la notation algorithmique (on pourra s'inspirer des travaux de Moss sur le domaine, cités au § 4.1.3.).

En "aval" de l'expression algorithmique, on pourrait envisager une activité de traduction vers un langage d'assemblage (dans le cadre d'un enseignement de compilation, pour la partie génération de code): par exemple vers un langage d'une machine à pile (cela ne pose pas de problèmes) ou vers, par exemple, le langage d'assemblage du 68000 mais dans ce cas l'expression de la traduction est plus complexe.

Il est envisageable d'illustrer un cours de compilation, pour la partie analyse et l'expression des sémantiques, en montrant l'intérieur du logiciel Idalgo, et en visualisant les arbres abstraits manipulés (figure 4.10, arbre abstrait de l'algorithme "compter les 'a'").

figure 4.10: consultation de l'arbre abstrait d'un algorithme



CONCLUSION

Un premier thème d'étude présenté dans cette thèse est l'expressivité des moyens de communication fournis aux intervenants en EAO. Nous préconisons l'usage de langages de spécification exécutables pour faciliter une appropriation par les enseignants et pour favoriser une conception méthodique des logiciels. Prolog a servi de support pour étayer cette hypothèse.

Nous avons constaté dans diverses expériences de formation et dans l'expérimentation FLE que Prolog se prête bien à une appropriation facile par des non informaticiens, pour peu qu'on le leur présente en rapport avec leur culture. Nous espérons retrouver cette caractéristique dans une validation future des logiciels que nous avons réalisés.

Nous avons exploité dans divers contextes la variété des approches procédurale, relationnelle et grammaire de Prolog. L'approche grammaire a été particulièrement précieuse dans notre contexte dominé par des objets très structurés.

Ceci conduit à penser qu'une bonne manière de rapprocher les moyens d'expression de la culture des auteurs est de les adapter aux objets pédagogiques qui modélisent la connaissance enseignée.

Il faut donc poursuivre cette étude en utilisant d'autres langages de haut niveau, et en s'intéressant à d'autres sujets que la spécification des analyses de réponses: par exemple, étudier l'apport d'un langage orienté objets avec des enseignants de sciences expérimentales, pour la modélisation de phénomènes à simuler; ou encore utiliser un langage fonctionnel, avec des enseignants de mathématiques...

Le deuxième thème d'étude présenté dans cette thèse concerne la production de logiciels pour l'enseignement.

Du point de vue de l'analyse pédagogique, nous avons mis en évidence deux lignes directrices: la première consiste en un repérage d'activités pédagogiques desquelles on déduit des modèles de dialogues pour spécifier les aspects dynamiques du logiciel. La seconde est fondée sur l'identification d'objets spécifiques au domaine enseigné. Dans les deux cas, nous avons vu comment, cette démarche était précieuse lors de la réalisation du logiciel: en effet elle permet une conception globale des outils nécessaires, et elle est parfaitement adaptée au souci de paramétrage, notamment en rapport à la question de l'appropriation des logiciels par les enseignants utilisateurs.

Du point de vue de l'approche de production, nous avons exploité ce qui précède dans un contexte de prototypage et de développement incrémental, classique dans d'autres domaines de l'informatique. Le prototypage est nécessaire du fait des difficultés de spécification des logiciels. L'enrichissement progressif porte sur les activités proposées et sur les formes de dialogues qui en découlent et sur les objets pédagogiques. D'autre part il concerne à la fois l'auteur, l'enseignant utilisateur, voire l'apprenant.

Du point de vue de l'architecture des systèmes de production, nous avons proposé un système comportant plusieurs niveaux. On trouve tout d'abord un environnement général de programmation, qui assure une grande puissance d'intervention (en principe pour un informaticien) et qui garantit une certaine portabilité aux logiciels produits. On trouve ensuite des fonctions générales EAO, rendues plus accessibles aux auteurs du fait des choix des moyens d'expression dont nous avons parlé plus haut. L'approche objet est présente dès ce niveau. Elle se prête bien à la démarche de prototypage et d'enrichissement progressif.

Nous avons présenté le troisième niveau comme spécifique du domaine

enseigné. Il s'avère que l'on peut préciser les choses un peu plus a posteriori, ici encore en référence aux objets pédagogiques manipulés: au delà du contexte d'enseignement de la programmation qui nous a amené à le définir, Tangram nous semble particulièrement adapté à tout domaine dans lequel les objets manipulés sont très structurés et dans la mesure où ces structures sont un élément important de ce qui est enseigné.

Le logiciel Idalgo a plus été un moyen d'étude des points qui précèdent qu'un objectif à part entière. Nous avons souligné les limites qu'il comporte à l'heure actuelle. Il nous semble qu'il constitue une bonne maquette sur laquelle on peut s'appuyer pour offrir aux étudiants un outil de travail intéressant: les activités proposées sont tout à fait pertinentes dans un contexte d'initiation à l'algorithmique, dans la mesure où elles renforcent une attitude systématique et où elles fournissent une bonne illustration de phénomènes dynamiques. Il faut sans doute se poser la question de la mise en oeuvre de tels outils dans le contexte traditionnel de l'enseignement. Par ailleurs, on peut s'interroger sur la démarche qu'il faut suivre pour produire de tels outils: faut-il développer des outils spécifiques pour l'enseignement, ou faut-il réutiliser, adapter des environnements de programmation existants?

Nous avons exploité largement les possibilités de Prolog en tant que langage de spécifications exécutable, et donc de réalisation de maquettes. La question reste de savoir s'il peut être maintenu comme langage de production en vraie grandeur de logiciels pour l'enseignement même diffusé sur des micro-ordinateur.

ANNEXE: le traitement séquentiel

Le deuxième thème de [ScP 88] est consacré aux structures itératives, le domaine d'illustration choisi étant le traitement de textes. Ce domaine est concrétisé par une modélisation de la manipulation des fichiers séquentiels: les exemples-types font référence à une machine fictive de consultation d'un texte, caractère par caractère, définie par la spécification d'actions nommées de modification de son état. Cette machine permet de remettre à plus tard les problèmes précis de terminaison en "marquant" la fin du texte par un caractère spécial. Les différents aspects abordés dans ce thème sont:
- *premier aspect*: le parcours d'une séquence (notion d'invariant et premier schéma de traitement séquentiel); l'exemple-type choisi est "le calcul de la longueur d'un texte donné". Les exercices proposés étant: "le calcul du nombre de 'A' d'un texte donné", "le calcul pour déterminer si le texte donné comporte plus de 'A' que de 'E'", etc .
- *deuxième aspect*: la recherche séquentielle; l'exemple-type choisi est "un texte donné comporte-t-il au moins un 'A' ?".
- *troisième aspect*: le parcours d'une sous-séquence (l'analyse s'appuie à la fois sur la notion de traitement séquentiel et sur la notion de recherche séquentielle); l'exemple-type choisi est: "déterminer si un texte comporte au moins x (entier positif donné) occurences d'un caractère c donné".
- *quatrième aspect*: démarche descendante dans l'analyse du traitement d'une séquence, fondée sur l'introduction d'une séquence intermédiaire (le modèle de manipulation des fichiers séquentiels devient modèle d'analyse des problèmes "itératifs"); l'exemple-type choisi est: "calcul du nombre de couples 'LE' dans un texte donné" (version sur la suite intermédiaire des couples de caractères).

Pour illustrer la démarche de construction d'algorithme appliquée, nous allons développer les étapes de construction des deux exemples-types choisis pour le premier et le quatrième aspect ci dessus.

Démarche de construction de l'exemple-type 1

Enoncé de l'exemple:

Etant donné un texte accessible à l'aide d'une machine-caractères de consultation, écrire un algorithme qui affiche la longueur (nombre de caractères) du texte. Par exemple, pour le texte "ceci est un exemple" le résultat est 19.

Il faut d'abord calculer la longueur du texte avant de l'afficher, on introduit donc une information intermédiaire, nommée longtexte et une action CompterCar. Pour spécifier cette action, nous désignons par lg(t) la longueur d'un texte t.

Un premier énoncé résume cette étape: Enoncé 1.

---

lexique
{T est le texte donné accessible par une machine-caractères}
longtexte: un entier ≥ 0
CompterCar: une action
{état initial: indifférent}
{état final: longtexte = lg(T)}
algorithme
CompterCar; écrire(longtexte)

Choix d'une classe de problème:

L'action CompterCar relève d'une classe générale de problèmes de

traitement de séquences: on parle de parcours d'une séquence marquée pour dire que l'on énumère tous les éléments de la séquence en appliquant un même traitement à chacun.

L'analyse d'un tel problème consiste à spécifier les composants de l'algorithme itératif: initialisation et progression du parcours; traitement de l'élément courant, initialisation et terminaison du traitement.

Définition de la séquence à traiter:

Dans cet exemple la séquence à traiter est la séquence des caractères du texte. Cette séquence est accessible par une machine-caractères, les actions correspondant à l'initialisation et à la progression du parcours sont donc: démarrer-caractère et avancer-caractère.

Analyse du traitement:

Il reste à analyser le traitement. Une définition récurrente du résultat attendu permet de choisir les variables et de trouver l'invariant de l'itération. L'exemple choisi permet d'illustrer une situation fréquente où l'on procède de la manière suivante:

- on explicite la fonction f qui à la séquence donnée S associe le résultat attendu. Dans l'exemple, il s'agit de la fonction lg.
- on introduit une variable v définie par l'invariant: v = f(pg) ; pg ou partie gauche étant la séquence de caractères situés à gauche du caractère courant. Dans l'exemple cette variable v sera nommée cpt.
- on décrit le traitement de l'élément courant et l'initialisation à partir d'une définition récurrente de la fonction f:

f([]) = valeur initiale

f(S.e) = g(e,f(S))

Dans cette définition [] désigne la séquence vide et S.e désigne la séquence obtenue en ajoutant un élément e à droite (à la fin) d'une séquence S.

De la première relation de la définition récurrente, on déduit l'initialisation: v <- valeur initiale et de la seconde, le traitement de l'élément courant EC: v <- g(EC,v).

Sur l'exemple, la définition récurrente de la fonction lg est:

lg([]) = O dont on déduit: cpt <- 0

lg(t.c)= lg(t) + 1 dont on déduit: cpt <- cpt + 1

Application d'un schéma:

Le schéma à appliquer dans l'exemple, correspond à la forme 1 du schéma de parcours d'une séquence marquée (avec traitement intégré de la séquence vide et du premier élément).

Schéma de parcours d'une séquence marquée (forme 1):

```
{état initial de l'algorithme}
    Démarrer
    Initialisation du traitement
    tantque EC ≠ marque {description de l'invariant}
        {EC désigne l'élément courant}
        Traitement de l'élément courant
        Avancer
    Terminaison du traitement
{état final de l'algorithme}
```

L'application de ce schéma dans le cas de l'exemple donne l'algorithme de l'action CompterCar (voir Enoncé 2).

Enoncé 2 (complétant l'énoncé 1):

```
CompterCar: l'action
    lexique {on utilise la machine-caractères}
        cpt: un entier ≥ 0
    algorithme
        {état initial: texte T disponible}
        démarrer-caractère
        cpt <- 0
        tantque caractère-courant ≠ marque {invariant: cpt = lg(pg)}
            cpt <- cpt +1
            avancer-caractère
        {caractère-courant = marque et cpt = lg(pg)}
        longtexte <- cpt
      {état final: longtexte = lg(T)}
```

Démarche de construction de l'exemple-type 2

Enonçé de l'exemple:

On considère un texte accessible par une machine-caractères. Ecrire un algorithme qui affiche le nombre de 'le' du texte. Par exemple, pour le texte: elle appelle le vieil employé de l'hôtel; le résultat est 3

Diverses manières de construire cet algorithme sont possibles en variant sur la nature de la séquence considérée au début de l'analyse. Nous développons cet exemple en raisonnant sur la séquence de couples de caractères consécutifs du texte donné.

Choix d'une classe de problème: Le problème posé correspond à un parcours d'une séquence.

Définition de la séquence à traiter: La séquence à traiter est la séquence de couples de caractères consécutifs du texte donné. Pour caractériser l'accés aux éléments il faut spécifier une machine-couples décrivant les actions démarrer-couple et avancer-couple correspondant à l'initialisation et à la progression du parcours et le prédicat couple-terminal correspondant à la marque de la séquence de couples.

Spécification de la machine-couples:

lexique:

couple: le type <p,s: des caractères>
couple-courant: un couple {couple courant}

démarrer-couple: une action
{état initial: indifférent}
{état final: couple-courant = premier couple, éventuellement terminal}

avancer-couple: une action
{état initial: couple-courant n'est pas terminal}
{état final: couple-courant = prochain couple éventuellement terminal}

couple-terminal: une fonction -> un booléen
{désigne vrai si le couple courant est terminal et faux sinon}

La relation entre la machine-couples et la machine-caractères est fixée en définissant la relation invariante entre caractère courant et couple courant:

(p de couple-courant = dernier(pg) ou '_' si pg = []) et
(s de couple-courant = caractère-courant)

D'où la réalisation de la machine-couples:

<u>Réalisation de la machine-couples</u> (complétant l'énoncé de spécification):

---

démarrer-couple: l'action
    démarrer-caractère; p de couple-courant <- '_';
    s de couple-courant <- caractère-courant

avancer-couple: l'action
    p de couple-courant <- s de couple-courant; avancer-caractère;
    s de couple-courant <- caractère-courant

couple-terminal: la fonction -> un booléen
    couple-terminal: (s de couple-courant = marque)

Cet énoncé peut être transformé, en explicitant le couple courant en termes du caractère courant puisque caractère-courant = s de couple-courant, et en introduisant une variable cp pour représenter p de couple-courant.

<u>Analyse du traitement</u>: Nous définissons la fonction nble qui à une séquence de couples donnée associe le nombre de couples 'LE'.

La définition récurrente de la fonction nble est:

nble([]) = 0
nble([e]) = 0
nble(S.e1.e2) = selon <e1,e2>
        <e1,e2> = <'L','E'>: nble(S.e1) + 1
        <e1,e2> ≠ <'L','E'>: nble(S.e2)

En introduisant la variable cptle définie par l'invariant: cptle = nble(pg), on déduit de la première relation de la définition récurrente, l'initialisation

cptle <- 0 et de la seconde, le traitement du couple courant:
si p de couple-courant = 'L' et s de couple-courant = 'E': cptle <- cple + 1

Application du schéma: Le schéma à appliquer correspond comme dans l'exemple-type précédent à la forme 1 du schéma de parcours d'une séquence marquée. L'application de ce schéma donne l'algorithme suivant:

Algorithme de nombre de 'LE' version couple de caractères consécutifs:

```
lexique:
    {le couple courant est représenté par <cp, caractère-courant>}
    cp: un caractère {cp = dernier(pg) ou cp ='_' si pg = []}
    cptle: un entier ≥ 0 {compteur des 'LE'}

algorithme:
    démarrer-caractère; cp <- '_'
    cptle <- 0
    tantque caractère-courant ≠ marque {cptle = nble(pg)}
        si cp = 'L' et caractère-courant = 'E': cptle <- cptle + 1
        cp <- caractère-courant; avancer-caractère
    écrire(cptle)
```

BIBLIOGRAPHIE

Classement par thèmes des ouvrages cités

1. Enseignement assisté par ordinateur

1.1] Logiciels d'enseignement

[PiB 87], [Muc 87], [Pap 73], [Pap 81]

1.2] Tuteurs intelligents

[SlB 82], [Cla 86], [Niv 87], [Wen 87], [Car 70], [Cla 79], [Cla 83], [FFB 84], [Hat 85], [Pal 86], [Cou 87], [Sch 87], [BBF 87], [Nic 87], [BBK 81]

1.3] Génie didacticiel: modèles de cycle de vie

[Fis 85], [Rec 85], [HBB 86], [Dem 86]

1.4] Génie didacticiel: outils de production

[Que 83], [For 85], [MPB85], [MPB 87], [Dia 88]

1.5] Génie didacticiel: exemples d'outils de production

[ADI 84], [Mor 86], [Irp 87], [DSN 87], [CoB 87], [Léo 87]

1.6] Travaux E.A.O. réalisés au L.G.I. en connection avec le projet

[Pai 81] ,[Ada 83], [Mos 83], [Sch 83], [Luc 83], [Luc 83b], [AbS 84], [Luc 84], [Luc 86a], [Luc 86b], [Luc 87]

2. Enseignement et E.A.O. de la programmation

2.1] L'E.A.O. de la programmation [BoS 87] [Bou 87]

[Gol 75], [KoB 75], [BBA 76], [BoC 86], [Mil 78], [Bar 79], [AdL 80], [Que 80], [Sha 82], [BrS 83], [BrS 84], [BFW 84], [Vos 84], [AnR 85], [Jos 85], [FDV 88], [Gra 88]

2.2] Travaux présentant le contexte d'enseignement de la programmation

[Sch 79], [LPS 83], [Sch 84], [ScP 88]

2.3] Travaux du L.G.I. en E.A.O. de la programmation

[Zam 84], [Agu 85], [Gue 85], [Lie 85], [AGP 85], [PeG 85], [Cag 86], [Luc 86a], [Luc 86b], [Gir 86], [Luc 87], [CGL 88] , [PCP 88] ,[GuP 88], [Gue 89]

3. Génie logiciel et intelligence artificielle

3.1] Génie logiciel et environnements de programmation

[Boe 81], [TeR 81], [MRR 82], [BCC 87], [Kra 87]

3.2] Prototypage

[Cho 85], [Bez 85], [RLT 85], [BHT 86], [KoM 86], [Leg 87], [Cho 88]

3.3] Développement de systèmes experts

[Dem 85], [Hat 86], [Mag 88]

3.4] Spécifications formelles des langages de programmation

[Bac 60], [Knu 68b], [LLS 68], [Liv 78], [MaW 79], [Pag 81], [Plo 81], [Kah 87]

4. Prolog

- ouvrage général sur le langage, ses applications et ses implantations: [Cnd 86]
- ouvrages conseillés: [StS 86], [CoC 88] actualisation et extension de [CCP 80]

4.1] Le langage Prolog

[CCP 80], [ClM 81], [CKC 83], [GKP 85], [Del 85], [Luc 85], [StS 86], [Bra 86], [CoC88]

4.2] Prolog et le traitement des langages de programmation

[Col 75], [War 80], [Sim 81], [Mos 81], [Mon 84], [Luc 87]

4.3] Prolog et le traitement des langues naturelles

[CKR 73], [Dah 77], [PeW 80], [Mac 82], [Per 83], [AbD 84], [Sai 85]

4.4] Prolog et bases de données

[Dah 77], [GaM 78], [GMN 82], [Gal 83], [NaW 83], [NgW 84], [AdN 84]

4.5] Autres domaines d'application

[War 74], [BBL 79], [Din 79], [Enn 81], [DLL 81], [BJM 82], [RLT 85], [Leg 87], [Sch 87], [Cou 87b]

4.6] Environnements Prolog

[WPP 79], [CMF 80], [Don 83], [BJM 83], [PPW 84], [PRO 85], [PRO 85b], [MiM 86], [Mha 87], [LPA 88], [PRO 89]

Références bibliographiques

[AbS 84] D. ABRY, M. SCHOLL
Informatique et Français Langue Etrangère: Aspects pédagogiques du projet Mosaïque.
Revue Le Français dans le Monde, n° 186, juillet 1984

[AbD 84] H. ABRAMSON, V. DAHL
Gapping grammars. Proceeding of the Second International Logic Programming Conference, Université d'Uppsala, Juillet 1984

[Ada 83] J.M. ADAM
Méthodes et outils pour la production de didacticiels: l'environnement informatique du projet Mosaïque.
Thèse de 3ème cycle INPGrenoble décembre 1983

[ADI 81] AGENCE DE L'INFORMATIQUE
Rapport E.A.O., Paris 1981

[ADI 83] AGENCE DE L'INFORMATIQUE
Système DIANE: Enseignement Assisté par Ordinateur, Paris 1983

[ADI 84] AGENCE DE L'INFORMATIQUE
Présentation générale de Diane, Paris 1984

[Adl 80] A. ADAM, J. LAURENT
A system to debug student programs.
Artificial Intelligence n°15, pp 75-122, 1980

[AdN 84] M. ADIBA, G.T. NGUYEN

Logic programming for a generalized data management system
Rapport de recherche Tigre n°12, Janvier 1984

[AGP 85] J.M. ADAM, V. GUERAUD, S. PAINVIN, J.P. PEYRIN,
G. FAFIOTTE, P.C. SCHOLL

Projet Didalp: production de didacticiels sous le système Diane
Rapport d'activités du groupe CARRY 7 ADI IMAGrenoble 1985

[Agu 85] J.L. AGUIRRE

Réalisation d'un didacticiel en vraie grandeur pour l'enseignement de l'algorithmique et de la programmation
Rapport de 3ème année de l'ENSIMAG INPGrenoble 1985

[AnR 85] J.R. ANDERSON, B.J. REISER

The Lisp Tutor, Byte 10 (4) 1985

[Ars 88] J. ARSAC

Des ordinateurs à l'informatique
Colloque sur l'histoire de l'informatique Grenoble 3-4-5 Mai 1988

[ASU 86] A.V. AHO, R. SETHI, J.D. ULLMAN

Compilers: Principles, Techniques, and Tools
Addison-Wesley Publishing compagny 1986

[Bac 60] J.W. BACKUS

The Syntax and Semantics cf the Proposed Int. Algebraic Language of the Zürich ACM-GAMA Conf. Information Processing
Proc. of The Int. Conf. on Inf. Processing,UNESCO Paris 1960

[Bar 79] J. BARRE
Système d'apprentissage des techniques élémentaires de compilation: Satec. Thèse de doctorat de 3ème cycle, Rennes I, juin 1979

[BBA 76] A. BARR, M. BEARD, R.C. ATKINSON
The computer as tutorial laboratory: the Stanford BIP project.
International Journal of Man-Machine Studies n°8, pp 567-595, 1976

[BBF 87] P. BROUAYE, E. BRUILLARD, E. FERRET, G. WEIDENFELD
Appat, un tuteur intelligent pour l'apprentissage des tableurs par la résolution de problèmes. EAO 87 Cap d'Agde mars 1987

[BBK 81] J.S. BROWN, R.R. BURTON, J. de KLEER
Pedagogical, Natural language and Knowledge Engineering Techniques in Sophie I, II and III.
Intelligent Tutoring Systems, Academic Press 1981

[BBL 79] A. BUNDY L. BYRD G. LUGER C. MELLISH R. MILNE M. PALMER. MECHO a program to solve mechanics problems
Working paper n° 50 1/104 Dept. of AI Univ. of Edinburgh 1979

[BCC 87] M. BIDOIT, F. CAPY, C. CHOPPY, N. CHOQUET, C. GRESSE, S. KAPLAN, F. SCHLIENGER, F. VOISIN
Asspro: un environnement de programmation interactif et integré.
Revue TSI 1987

[Ben 84] M. BENNANI
Adjonction d'entrées-sorties vocales en E.A.O.: amélioration du dialogue et nouveaux champs d'application
Thèse de 3ème cycle Université de Nancy I 1984

[BeF 82] H. BESTOUGEFF, J.P. FARGETTE
Enseignement et ordinateur. Cedic/Nathan 1982

[Bez 85] J. BEZIVIN
Smalltalk et le prototypage. Revue Génie Logiciel n°3 1985

[BFW 84] E. BRUILLARD, E. FERRET, M. WEIDENFELD, G. WEIDENFELD. Apilog: connaitre et pratiquer Prolog. Développé par Softia et édité par Cedic/Nathan

[BHT 86] A. BONNET, J.P. HATON, J.M. TRUONG-NGOC
Systèmes Experts. Vers la maîtrise technique. InterEditions 1986

[BJM 82] G. BARBERYE, T. JOUBERT, M.MARTIN
Oasis: un outil d'aide à la spécification interactive et structurée
Proc. Journées Bigre Grenoble 1982

[BMQ 84] A. BOUDJOGHRA, A. MEDVEDEFF, M. QUERE
E.A.O: vers un partenaire de résolution de problèmes utilisant plusieurs experts. Le projet MEDIAN. RR 84-R-052 CRINancy 1984

[BoC 86] J. BONAR, R. CUNNINGHAM
Bridge: an intellignet tutor for thinking about programming
Proceeding of the ICAI Research Workshop, Windermere 1986

[Boe 81] B.W. BOEHM
Software Engineering Economics.
Prentice Hall, Englewood Cliffs, NJ 1981

[Bon 80] A. BONNET
De l'application de l'intelligence arificielle à l'enseignement à l'enseignement assisté par ordinateur: l'E.I.A.O. Caen Sept 1980

[Bon 84] A. BONNET
L'Intelligence Artificielle: promesses et réalités. InterEditions 1984

[Bos 87] B. du BOULAY, C. SOTHCOTT
Computers Teaching Programming: An Introductory Survey of the Field. Artificial intelligence and education, volume one: Learning environments and Tutoring systems
edited by R. Lawler et M. Yazdani, Ablex 1987

[Bou 87] B. du BOULAY
Intelligent systems for teaching programming in artificial intelligence tools in education
edited by P. Ercoli and R. Lewiw IFIP north-holland 1987

[Bra 86] I. BRATKO
PROLOG programming for artificial intelligence
International Computer Science Series, Addison-Wesley, 1986

[BrS 83] M.H. BROWN, R. SEDGEWICK
Brown University Instructional Computing Laboratory
Technical Report N° CS-83-28, Brown University, December 1983

[BrS 85] M.H. BROWN, R. SEDGEWICK
A system for algorithm animation. Dept. of Computer Science Brown University. Technical Report N° CS-84-01 January 1984

[Cag 86] J.M. CAGNAT
Présentation du projet Arcade
Rapport interne IMAGrenoble L.G.I septembre 1986

[Cag 88] J.M. CAGNAT
Meccano de tris: une boîte d'outils pour trier des cubes
Rapport ARCADE n° 11, I.M.A.G L.G.I. Grenoble, Août 1988

[Car 70] J.R. CARBONELL
AI in CAI: an artificial intelligence approach to computer-assisted instruction. IEEE Transactions on man-machine systems, vol.11, n°4. 1970

[Cas 84] Documentation CASSIE
Diane Arlequin: manuel d'utilisation

[CCP 80] H. COELHO, J.C.COTTA, L.M. PEREIRA
How to solve it with Prolog
Laboratoire National de Génie Civil, Lisboa, Portugal, 1980

[CDD 85] D. CLEMENT, J. DESPEYROUX, T. DESPEYROUX, L. HASCOET, G. KAHN. Natural semantics on the computer
Rapport de recherche INRIA Sophia Antipolis juin 1985

[CGL 88] J.M.CAGNAT, V. GUERAUD, I. LIEM, S. PAINVIN, J.P. PEYRIN. Un laboratoire pour l'enseignement de la programmation. Colloque sur l'évolution de l'outil informatique à l'Université. 14-16 septembre 1988 - Poitiers

[CGV 80] P.Y. CUNIN, M. GRIFFITHS, J. VOIRON
Comprendre la compilation, Springer-Verlag 1980

[Cho 57] N. CHOMSKY
Syntactic Structures Mouton et co La Haye 1957
(traduction française: Structures syntaxiques Le Seuil 1969)

[Cho 59] N. CHOMSKY
On certain Formal Properties of Grammars,
Information and Control, 2 1959 pp 137-167

[Cho 85] C. CHOPPY
Techniques et aspects du prototypage - Revue Génie Logiciel n°3 1985

[Cho 88] C. CHOPPY
Maquettage et prototypage: panorama des outils et des techniques
Revue Génie Logiciel et Systèmes Experts n°11 Mars 1988

[CKR 73] A. COLMERAUER, H. KANOUI, P. ROUSSEL, R. PASERO
Un système de communication homme-machine en français
GIA, UER de Luminy, Université d'Aix-Marseille, 1973

[CKV 83] A. COLMERAUER, H. KANOUI, M. VAN CANEGHEM
Prolog, bases théoriques et développements actuels
TSI Vol.2 n° 4 juillet-août 1983 pp 271-311

[Cla 79] W.J.CLANCEY
Transfer of rule-based expertise through a tutorial dialogue
Doctoral dissertation Stanford University California 1979

[Cla 83] W.J.CLANCEY
Guidon.
Journal of Computer-Based Instruction, vol. 10, n°1, pp. 8-14 1983

[Cla 86] W.J.CLANCEY
Intelligent Tutoring Systems: A Tutorial Survey
Dept of Computer Science Stanford September 1986

[ClM 81] W.F. CLOCKSIN, C.S. MELLISH
Programming in Prolog. Springer Verlag, Berlin, 1981
(traduction française: Programmer en Prolog, Editions Eyrolles 1985)

[CMF 80] W. CLOCKSIN, C. MELLISH, R. FISHER
The RT.11 Prolog System
Software Report 5a, DAI University of Edinburgh, December 1980

[Cnd 86] M. CONDILLAC (nom collectif du groupe Prolog de l'Afcet)
Prolog, fondements et applications
Afcet informatique Dunod informatique 1986

[CoB 87] Course Builder

Course Builder Manual. TeleRobotics International, Inc., 1987

[CoC 88] H. COELHO, J.C. COTTA

Prolog by example: how to learn, teach and use it.
Springer-Verlag 1988

[CoG 86] M. COURANT G. GUEVEL

Validation d'une base de connaissances hybride avec objets et règles de production. Rapport INRIA Rennes Irisa Novembre 1986

[CoK 79] D. COULON, D. KAYSER

Modalités du dialogue en enseignement assisté par ordinateur
IRIA Sesori Arc et Sénans 21-23 mai 1979

[Col 70] A. COLMERAUER

Les systèmes Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Publication interne n°43. Dept d'Informatique. Université de Montréal. Sept 1970

[Col 75] A. COLMERAUER

Les grammaires de métamorphose. Natural Language Communication with Computers, Springer Verlag 1975

[Cou 87] B. COULETTE

Iphigenie: un didacticiel expert en methodologie de developpement de projets logiciels. EAO 87 Cap d'Agde mars 1987

[Cou 87b] M. COURANT

La compréhension de petites annonces dans le système-expert Havane

Thèse de l'Université de Rennes I, Novembre 1987

[Dah 77] V. DAHL

Un système déductif d'interrogation de banques de données en espagnol.

Thèse UER de Luminy, Université d'Aix-Marseille II, Novembre 1977

[DeA 82] C. DELOBEL, M. ADIBA

Bases de données et systèmes relationnels

Dunod Informatique Bordas Paris 1982

[Del 85] J.P. DELAHAYE

Outils logiques pour l'intelligence artificielle

Editions Eyrolles décembre 1985

[Dem 85] L. DEMIANS D'ARCHIMBAUD

Panorama des outils de génie logiciel pour le développement des applications de l'intelligence artificielle.

Revue Génie logiciel n° 2 1985

[DeM 88] P. DERANSART, J. MALUSZYNSKI

A grammatical view of logic programming

Rapport de recherche INRIA Rocquencourt, Aout 1988

[Dem 86] F. DEMAIZIERE

L'enseignement assisté par ordinateur. OPHRYS 1986

[Des 84] T. DESPEYROUX
Executable Specification of Static Semantics, Semantics of Data Types
Lecture Notes in Computer Science, Vol. 173, June 1984

[Des 86] T. DESPEYROUX
The Typol Manual. INRIA Sophia Antipolis décembre 1986

[Dia 88] B. DIAWARA
Etude comparative des langages auteurs Duo, Arlequin et DrLéo
TSI vol 7 n°1 1988

[Dij 76] E.W. DIJKSTRA
A discipline of programming. Prentice Hall 1976

[Din 79] M. DINCBAS
Prolog et un exemple de système expert écrit en Prolog
Doc. CERT 2/3122 1979

[DKL 83] V. DONZEAU-GOUGE, G. KAHN, B. LANG, B.MELESE
Outline of a tool for document manipulation
IFIP Paris septembre 1983

[DLL 81] Ph. DONZ, Ph. LEBAUDE, B. LEPAPE
Foll, un outil logiciel pour les outils génie logiciel
Proc. Journées Bigre 1981

[Don 83] Ph. DONZ
Foll, une extension au langage Prolog
CRISS, Université des sciences sociales de Grenoble, Novembre 1983

[DSN 87] D. DUMOULIN, A. SCHMITZ, J.D. NICOUD
Prof: un système auteur interactif
EAO 87 Cap d'Agde 23-24-25 mars 1987

[DuT 72] O. DUCROT, T. TODOROV
Dictionnaire encyclopédique des sciences du langage
Points Sciences Humaines Editions du Seuil 1972

[Enn 81] J.R. ENNALS
Logic as a computer language for children
Educational Computing October 1981

[Eur 85] Documentation EUROFORMATIQUE
Diane: l'éditeur fonctionnel

[FaQ 84] C. FAY, M. QUERE
Vers l'enseignement intelligement assisté par ordinateur: comprendre, résoudre et expliquer un problème.
Rapport de Recherche 84-R-010 CRINancy 1984

[FDV 88] I. FERNANDEZ DE CASTRO, A DIAZ DE LIARRAZA SANCHEZ, F VERDEJO MAILLO. A cooperative architecture for tutoring tasks. Journées Systèmes Experts Avignon 1988

[FFB 84] D. FIESCHI, M. FIESCHI, G. BOTTI, M. JOUBERT
Un système expert d'aide à l'enseignement médical.
EAO84 Lyon 1984

[Fis 85] J. FISZER
Elaboration de didacticiels: étapes, problèmes, difficultés
Bulletin EPI n°39 1985

[For 85] E. FORTE
Les langages-auteur d'E.A.O: pour qui et pour quoi faire ?
Laboratoire de Microinformatique E.P.F.Lausanne Décembre 1985

[Gal 83] H. GALLAIRE
Prolog et bases de données.
Journées Prolog, CNET, Perros-guirec, Mars 1983

[GaM 78] H. GALLAIRE, J. MINKER
Logic and databases - Plenum press, New-York, 1978

[Gir 86] X. GIROD
Etude d'un environnement de travaux-pratiques en enseignement de la programmation: techniques de transformations de programmes récursifs en itératifs. Rapport de D.E.A. INPGrenoble septembre 1986

[GKP 85] F. GIANNESINI, H. KANOUI, R. PASERO, M. Van CANEGHEM. Prolog - InterEditions, 1985

[GMN 82] H. GALLAIRE, J. MINKER, J.M. NICOLAS
Logic and databases: an overview and survey
joint report CERT / CGE / univ. of Maryland october 1982

[Gol 75] I.P. GOLDSTEIN

Summary of Mycroft: a system for understanding simple picture programs. Artificial Intelligence n° 6, pp 249-288, 1975

[Gra 86] Anna GRAM (groupe de travail A.F.C.E.T.)

Raisonner pour programmer - Dunod 1986

[Gra 88] M. GRANDBASTIEN

Une approche à base de connaissances pour l'enseignement de la programmation. Conception et réalisation de Saida: système d'aide à l'implantation de données abstraites
Thèse de doctorat d'état Université de Nancy I juillet 1988

[Gri 81] D. GRIES

The science of programming. Springer-Verlag Berlin 1981

[Gue 85] V. GUERAUD

Etude des fonctions E.A.O. nécessaires pour un didacticiel de soutien à l'enseignement de la récursivité: réponses du système Macintosh à des besoins E.A.O. Rapport de D.E.A. I.N.P.Grenoble juin 1985

[Gue 89] V. GUERAUD

Un jeu de rôles dans le laboratoire ARCADE: une autre façon d'enseigner la programmation.
Thèse de l'Université J. Fourier, Grenoble, 1989

[GuP 88] V. GUERAUD, J.P. PEYRIN

Un jeu de rôle pour l'enseignement de la programmation
Colloque francophone sur la didactique de l'informatique Paris 1988

[Hat 85] M.C. HATON, J.P. HATON

Application de l'intelligence artificielle dans l'enseignement des langues. Rapport de Recherche CRINancy N°102 1985

[Hat 86] J.P. HATON

Methodes et outils pour le développement de systèmes experts

Rapport de Recherche CRINancy 1986

[HBB 86] M. HENNART, J.F. BERTHON, M. BINSE

Ecrire des logiciels pédagogiques

Savoir&savoir faire informatique Cedic/Nathan 1986

[Hul 83] J.M. HULLOT

A multi-formalism programming environment IFIP Paris sept 1983

[Irp 87] IRPEACS

Système auteur de l'IRPEACS. Lyon 1987

[Jac 74] M. JACQUES

Le système OPE. Acta Electronica. Vol. 17. 1974

[Jer 81] P. JEROME

La simulation dans la pédagogie des sciences expérimentales

3ème conférence mondiale sur l'éducation. WCCE 81. Lausanne 1981

[JoS 85] W.L. JOHNSON, E. SOLOWAY

Proust. Byte 10 (4) 1985

[Kah 87] G. KAHN

Natural semantics

Rapport de recherche n° 601, INRIA Sophia-Antipolis, février 1987

[Knu 68a] D.E. KNUTH

The art of computer programming Vol. 1. Addison Wesley 1968

[Knu 68b] D.E. KNUTH

Semantics of context free languages

a- Math-Systems Theory Vol. 2 n°2 pp 127-145 1968

b- Math-Systems Theory Vol. 5 n°1 pp 95-96 1971

[KoB 75] E.B. KOFFMAN, S.E. BLOUNT

Artificial intelligence and automatic programming in CAI

Artificial Intelligence n°6, pp 215-234, 1975

[KoE] R.A. KOWALSKI, M.H. Van EMDEN

The semantics of predicate logic as a programming language.

JACM 23 n°4 oct 1976

[KoM 86]J. KOMOROWSKI, J. MALUSZYNSKI

Logic programming and rapid prototyping, Research Report

LITH-IDA-R-86-20, dept of Computer and information science,

Linkoping University, 1986

[Kow 79] R. KOWALSKI

Logic for Problem Solving. North Holland, New-York, 1979

[Kra 87] S. KRAKOWIAK
Rapport sur la formation en génie logiciel.
Revue Génie Logiciel n°8 Juin 1987

[Lef 84] J.M. LEFEVRE
Guide pratique de l'enseignement assisté par ordinateur
CEDIC/NATHAN 1984

[Leg 87] B. LEGEARD
Prototypage de logiciels avec le langage Prolog: méthodes et outils
Thèse de doctorat de l'INSA de Lyon Octobre 1987

[Lel 87] R. LELOUCHE
Apports de l'E.I.A.O. à l'E.A.O. EAO 87 Cap d'Agde mars 1987

[Léo 87] Dr. Léo version 2.0
Système-auteur pour ordinateurs macintosh. Apigraph 1987

[Lie 85] I. LIEM
Etude des fonctions E.A.O. nécessaires pour un didacticiel de soutien à l'enseignement des structures de données: cas de la structure séquence
Rapport de D.E.A. I.N.P.Grenoble juin 1985

[LiG 87] B. LISKOV, J. GUTTAG
Abstraction and Specification in Program Development
The MIT Press McGraw-Hill Book Company 1987

[LiP 88] I. LIEM, J.P. PEYRIN. Le logiciel Tris Internes.
Rapport Arcade n°7, I.M.A.G L.G.I. Grenoble février 1988

[Liv 78] C. LIVERCY
Théorie des programmes: schémas, preuves, sémantique
Dunod informatique 1978

[LKS 84] B. LANG, G. KAHN, P. SIMON, Ph. LUDEAU, G. LATGE
L'E.A.O.: un domaine d'application du génie logiciel ?
E.A.O 84 Lyon 4-5 septembre 1984

[LLS 68] P. LAUER, P.LUCAS, H. STIGLEITNER
Method and notation for the formal definition of programming languages. Technical Report TR 25.087. IBM Laboratory Vienna 1968

[LPA 88] LPA Mac Prolog
Compilateur version 2, diffusé par Intellia Paris,
produit par Logic Programming Associates (GB)

[LPS 83] M. LUCAS, J.P. PEYRIN, P.C. SCHOLL
Algorithmique et représentation des données, tome1:
files, automates d'états finis. Masson manuels informatique 1983

[Luc 83a] A. LUCCI
Programmation en logique et E.A.O.: une expérience en E.A.O. du Français-Langue étrangère.
Rapport de D.E.A. de L'I.N.P.Grenoble 1983

[Luc 83b] A. LUCCI
Le système Voyelles. Rapport interne I.N.P.Grenoble septembre 1983

[Luc 84] A. LUCCI
Programmation en logique et E.A.O.: une expérience en E.A.O. du Français-Langue étrangère. E.A.O 84 Lyon 4-5 septembre 1984

[Luc 85] A. LUCCI
Différentes approches du langage Prolog: approches déclarative, procédurale et approche grammaire.
Support de cours, Orogestion Formation-Conseils,Grenoble 1985

[Luc 86a] A. LUCCI
Programmation en logique et E.A.O: une expérience en E.A.O. de la programmation. Séminaire SE et E.A.O. Pont-à-Mousson, janv 1986

[Luc 86b] A. LUCCI
Prolog et la production de logiciels d'enseignement.
Université d'été des Inspecteurs Généraux de l'Education Nationale
Saint Pierre de Chandieu 8 au 12 septembre 1986

[Luc 87] A. LUCCI
Prolog, Grammaires et E.A.O. E.A.O.87 Congrès francophone sur l'E.A.O. Cap d'Agde 23-24-25 mars 1987

[Mac 82] M. MAC CORD
Using Slots and Modifiers in Logic Grammars for Natural Language
Artificial Intelligence, Vol. 18, 1982

[Mae 84] E. MAERTENS
Les systèmes E.A.O. Actes du Forum E.A.O. 84. Lyon, sept 1984

[Mag 88] B. MAGUIRE
In incremental approach to expert systems development
Les systèmes experts et leurs applications Avignon 1988

[MaW 79] O.L. MADSEN, D.A. WATT
Extended Attribute Grammars. DAIMI PB. 105 1979

[MBP 85] F. MADAULE, P. BARRIL, B. de la PASSADIERE,
F. LE CALVEZ, M. POC, M. URTASIN
Quels outils informatiques pour réaliser des didacticiels ?
Rapport MASI n° 93, Université P. et M. CURIE, Paris, Juin 1985

[MBP 87] F. MADAULE, P. BARRIL, B. de la PASSADIERE,
F. LE CALVEZ, M. POC, M. URTASIN. Systèmes d'enseignement assisté par ordinateur: étude comparative. TSI vol 6 n° 1 1987

[Mel 86] B. MELESE
Mentor-V6, The Mentol programming language.
Sema-Metra juin 1986

[Mha 87] A. EL MHAMEDI
Interface Prolog-Graphique.
CNET Séminaire de programmation en logique 1986

[Mil 78] M.L. MILLER
A structured planning and debugging environment for elementary programming. International Journal Man-Machine Studies n°11, 1978

[MiM 86] A. MICHARD, E. MONCEYRON
Le système graphique Ash-Prolog et son utilisation pour le prototypage rapide d'interfaces homme-machine
Rapport Technique INRIA Sophia Antipolis Octobre 1986

[Mon 84] J.F. MONIN
Ecriture d'un compilateur réel en Prolog. Actes du séminaire sur la programmation en logique, CNET Lannion 1984

[Mor 83] P. MORAT
Une étude sur la base de la programmation algorithmique: notation et environnement de travail
Thèse de 3ème cycle, I.N.P.Grenoble, Décembre 1983

[Mor 86] J. MORINET-LAMBERT
Intégration de l'image en E.A.O. : l'illustrateur
Thèse de 3ème cycle, Université de Nancy 1, Octobre 1986

[Mos 81] C.D.S. MOSS. The formal description of programming languages using predicate logic - Ph. D. Degree Dept. of computing Imperial College, London, July 1981

[Mos 83] Equipe du projet MOSAIQUE
Création et mise en oeuvre de méthodes et d'outils informatiques d'aide à l'enseignement. Expérimentation sur un projet d'enseignement du Français langue étrangère. Rapport final du projet Mosaïque ADI IMAG / ULLG - CUEF, Grenoble, Octobre 1983

[MRR 82] J. MOSSIERE, J. RAYMOND, Y.ROUZAUD
Représentation interne et manipulation des programmes dans ADELE
Colloque Génie Logiciel AFCET Paris janvier 1982

[Muc 87] A. MUCCHIELLI
L'enseignement par ordinateur - Que sais-je ? P.U.F. 1987

[Nai 83] L. NAISH
MU-PROLOG 3.0: référence Manual,
University of Melbourne, Australie 1983

[NAW 83] J.C. NEVES, S.O. ANDERSON, H. WILLIAM
A Prolog implementation of Query-By-Example
Proceedings 7th International computing symposium, Nurnberg 1983

[NgW 84] G.T. NGUYEN, P. WINNINGER
Prolog et bases de données relationnelles
Rapport de Recherche Tigre n°25, L.G.I. Grenoble, Décembre 1984

[Nic 86] J.F. NICAUD
Intelligence Artificielle et Enseignement par Ordinateur
Séminaire SE et EAO Pont à Mousson. Janvier 1986

[Nic 87] J.F. NICAUD
Aplusix: un système expert en factorisation pour un logiciel d'E.I.A.O.
EAO 87 Cap d'Agde mars 1987

[NiV 87] J.F. NICAUD, M. VIVET
Les tuteurs intelligents: réalisations et tendances de recherches
Rapport de Recherche LRI n°359 juin 1987

[Pai 79] C. PAIR
La construction de programmes,
RAIRO Informatique, Vol 13, 2, 1979, p. 113-118

[Pai 81] S. PAINVIN
Projet Mosaique: cycle de préparation.
Compte-rendu Grenoble mars 1981

[Pag 81] F.G. PAGAN
Formal description of programming languages: a panoramic primer.
Prentice-Hall INC NewJersey 1981

[Pal 86] O. PALIES
L'I.A. au service de la formation: modèles et exemples
Université d'été Lyon, septembre 1986

[PaM 75] C. PAIR, J. MAROLDT
Introduction à une méthode de programmation déductive,
I.N.P.L, Nancy, 1975

[Pap 73] S. PAPERT
Uses of technology to enhance education
Memo-Logo n°8 Aim 298 Massachusetts Institute of Technology 1973

[Pap 81] S. PAPERT
Jaillissement de l'esprit: ordinateurs et apprentissage.
Flammarion 1981

[Pat 87] V. PATUREAU
Former des auteurs et des réalisateurs de didacticiels, un enjeu pour le marché de l'E.A.O.
E.A.O.87 Congrés francophone sur l'E.A.O. Cap d'Agde, Mars 1987

[PCP 88] S. PAINVIN, J.M. CAGNAT, J.P. PEYRIN, I. LIEM
Pour quelques approches de plus... Bulletin EPI n° 52, Nov 1988

[PeG 85] J.P. PEYRIN, V. GUERAUD
Propositions pour la spécification d'un didacticiel de soutien à l'enseignement de la programmation
Rapport de recherche n°586 IMAGrenoble LGI Décembre 1985

[Per 83] F. PEREIRA
Logic for Natural Language Analysis.
Technical note 275, SRI, Int. Californie, 1983

[PeW 80] F. PEREIRA, H.D. WARREN
Definite Clause Grammars for Language Analysis: a Survey of the Formalism and Comparaison with Augmented Transition Networks
Artificial Intelligence, Vol. 13, n°3, 1980

[PiB 87] M. PICARD, G. BRAUN
Les logiciels éducatifs. Que sais-je ? P.U.F. 1987

[Pla 82] Control Data PLATO system overview 1982

[Plo 81] G.D. PLOTKIN
A structural approach to operational semantics. DAIMI FN-19, Computer Science Department, Aarhus University, Sept 81

[PPW 84]L. PEREIRA, F. PEREIRA, D. WARREN, B. BOWEN, L. BYRD. C-Prolog User's Manual
Edited by F. PEREIRA SRI International Menlo Park California 1984

[PRO 85] PROLOGIA. Prolog II, Version 2.2, Manuel de référence, 1985

[PRO 85b] PROLOGIA. Prolog II, Version 2.2, Macintosh, 1985

[PRO 87] PROLOGIA. Prolog II, Version 2.4, Manuel d'utilisation sur Macintosh, 1987

[PRO 87] PROLOGIA. Compilateur Prolog II+, Macintosh 1989

[Que 80] M. QUERE
Contribution à l'amélioration des processus d'enseignement, d'apprentissage et d'organisation de l'éducation. L'ordinateur outil et objet de formation. Application au projet satire
Thèse de doctorat d'état Université de Nancy I Janvier 1980

[Que 83] M. QUERE
Outils pour l'enseignement assisté par ordinateur
Revue EPI Septembre 1983

[Rec 85] Groupe de travail RECODIC
Informatique et enseignement des sciences physiques.Guide de la conception et de l'évaluation de séquences éducatives utilisant l'ordinateur. 1985

[RLT 85] M. RUEHER, B. LEGEARD, B. TRAVERSAT
Prolog et le prototypage. Revue Génie Logiciel n°3 1985

[Rob 65] U.A. ROBINSON
A machine oriented logic based on resolution principle
JACM Vol 12, pp 23-41, 1965

[Rou 75] Ph. ROUSSEL
PROLOG manuel de référence et d'utilisation. G.I.A.
Université d'Aix-Marseille II, 1975

[Sai 85] P. SAINT-DIZIER
Handling Quantifiers Scoping Ambiguities in a Semantic Representation of Natural Sentences. Proceeding of the first workshop on Natural Language Understanding and Logic Programming. V. Dahl and P. Saint-Dizier Ed., 85

[Sav 87] J. SAVOY
Le livre electronique EBOOK3. E.A.O. 87 Cap d'Agde 1987

[Sch 79] P.C. SCHOLL
Vers une programmation systématique: étude de quelques méthodes, techniques et outils.
Thèse de doctorat d'état Université de Grenoble I 1979

[Sch 83] M. SCHOLL
Etude de méthodes en enseignement assisté par ordinateur du français langue étrangère: Résultats d'une expérience de production de didacticiel. Rapport de D.E.A CUEF IMSS Université des sciences sociales de Grenoble, octobre 1983

[Sch 84] P.C. SCHOLL
Algorithmique et représentation des données, tome 3: récursivité et arbres. Masson 1984

[Sch 87] C.B. SCHWIND
Un système expert pour l'E.A.O. des langues étrangères
E.A.O. 87 Cap d'Agde 1987

[ScP 88] P.C. SCHOLL, J.P. PEYRIN
Schémas algorithmiques fondamentaux: séquences et itérations
Masson 1988

[Sha 82] E.Y. SHAPIRO
Algorithmic program debugging, MIT Press, Cambridge MA, 1982

[Sho 76] E.H. SHORTLIFFE
Computer-based medical consultations: Mycin
American Elsevier Publishers, New York 1976

[Sim 81] M. SIMONET
W-Grammaires et logique du premier ordre pour la définition et l'implantation des langages
Thèse d'état U.S.M.Grenoble, Juillet 1981

[SlB 82] D. SLEEMAN, J.S. BROWN
Intelligent tutoring systems. Academic Press New York 1982

[Sol 86] E. SOLOWAY
Learning to program = learning to construct mechanisms and explanations. CACM, Volume 29, Number 9, September 1986

[StS 86] L. STERLING, E. SHAPIRO
The art of Prolog
MIT Press, Series in Logic Programming, Cambridge, USA, 1986

[TeR 81] T. TEITELBAUM, T. RESP
The Cornell Program Synthesizer: A syntax directed programming environment. CACM vol. 24 n° 9 pp. 563-573, September 1981

[Vos 84] G.M. VOSE
Macintosh Pascal - BYTE 9.6, 1984

[War 74] D.H. WARREN
Warplan: a system for generating plans
Memo n° 76 Dept. of Computational Logic Univ. of Edinburgh, 1974

[War 80] D.H. WARREN
Logic Programming and Compiler Writing
DAI Univ. of Edinburgh
Software practice and experience vol. 10, 1980

[Wen 87] E. WENGER
Artificial intelligence and tutoring systems: computational and cognitive approaches to the communication of knowledge
M. Kaufmann Publishers, inc. California, 1987

[Wer 85] H. WERTZ
Intelligence artificielle, application à l'analyse de programmes
Masson 1985

[WPP 79]D. WARREN, F. PEREIRA, L.M. PEREIRA
User's guide to DEC-system-10 Prolog
Dept. of Artificial Intelligence, University of Edinburgh 1979

[Zam 84] J. ZAMBRANO
E.A.O. et enseignement de la programmation: une maquette de didacticiel. Thèse de 3ème cycle I.N.P.Grenoble, Octobre 1984

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 Avril 1974

VU le rapport de présentation de Monsieur P.C. SCHOLL, Professeur

Monsieur LUCCI Alain

est autorisé à présenter une thèse en soutenance en vue de l'obtention du titre de DOCTEUR de TROISIEME CYCLE, spécialité " Informatique".

Fait à Grenoble, le 13 Février 1989

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble