Université de NANCY 1

Centre de Recherche en Informatique de Nancy

UNE APPROCHE A BASE DE CONNAISSANCES POUR L'ENSEIGNEMENT DE LA PROGRAMMATION

Conception et Réalisation de SAIDA : Système d'Aide à l'Implantation de Données Abstraites

THESE

présentée pour l'obtention
du doctorat d'Etat ès sciences
(informatique)

par

Monique GRANDBASTIEN

Soutenue le 9 Juillet 1988

devant le jury composé de:

Claude PAIR	Président et rapporteur
Jean-Pierre FINANCE	
Jean-Paul HATON	
Jean-Pierre LAURENT	
Maryse QUERE	
Pierre-Claude SCHOLL	Rapporteur
Martial VIVET	

Université de NANCY 1 Centre de Recherche en Informatique de Nancy

UNE APPROCHE A BASE DE CONNAISSANCES POUR L'ENSEIGNEMENT DE LA PROGRAMMATION

Conception et Réalisation de SAIDA : Système d'Aide à l'Implantation de Données Abstraites

THESE

présentée pour l'obtention
du doctorat d'Etat ès sciences
(informatique)

par

Monique GRANDBASTIEN

Soutenue le 9 Juillet 1988

devant le jury composé de:

Claude PAIR	Président et rapporteur
Jean-Pierre FINANCE	
Jean-Paul HATON	
Jean-Pierre LAURENT	
Maryse QUERE	
Pierre-Claude SCHOLL	Rapporteur
Martial VIVET	

Claude Pair me fait l'honneur de présider ce jury ; il a exercé son rôle de rapporteur en alliant une rigueur scientifique et des encouragements amicaux dont je le remercie vivement ; les nombreuses remarques qu'il m'a faites sur des versions successives des chapitres de cette thèse ont permis d'en améliorer la présentation, je lui en suis très reconnaissante.

Jean-Pierre Finance a accepté ce sujet qui dépassait largement les préoccupations de l'équipe programmation ; il a toujours su trouver des créneaux dans un emploi du temps très serré pour suivre l'avancement de mon travail et me faire de précieuses suggestions ; qu'il trouve ici l'expression de mes remerciements amicaux.

Pierre-Claude Scholl a lui aussi pris connaissance de versions successives du manuscrit malgré des lenteurs de transmission postale et ne m'a pas tenu rigueur des délais trop brefs que je lui imposais ; je le remercie pour l'intérêt qu'il a porté à ce travail, pour les discussions trop courtes que nous avons eues à son sujet, pour les suggestions qu'il m'a faites et que, j'espère, nous pourrons approfondir par la suite.

Jean-Pierre Laurent et Martial Vivet sont issus comme moi de l'équipe d'intelligence artificielle de J. Pitrat ; nous avons souvent travaillé depuis plus de quinze ans dans des cadres communs. Jean-Pierre anime maintenant le pôle 4 du PRC Intelligence Artificielle ; la qualité des discussions et l'ambiance amicale de travail des différents séminaires qu'il a suscités sont pour beaucoup dans l'avancement de ma propre réflexion. Martial partage avec moi depuis longtemps le goût des applications pédagogiques de l'informatique ; il a passé aussi beaucoup de temps dans les couloirs du 107 rue de Grenelle et développe actuellement des recherches sur les tuteurs intelligents. C'est un grand plaisir pour moi de les voir tous deux siéger à ce jury, ils ont accepté pour cela un long déplacement, je leur dis mon amicale gratitude.

Maryse Quéré et Jean-Paul Haton sont nancéiens. Maryse est probablement à l'origine de mes activités actuelles dans le domaine de l'enseignement puisque c'est avec elle que j'ai commencé à travailler à l'IREM en 1974 ; depuis, nous avons souvent collaboré à des actions de formation et de recherche ; je n'oublie pas que les projets de systèmes experts pour l'enseignement issus de l'appel d'offres de l'agence de l'informatique ont finalement vu le jour grâce à sa ténacité et je pense que notre coopération va se renforcer dès son retour au laboratoire. Jean-Paul partage mon intérêt pour l'intelligence artificielle ; il a créé et dirige l' équipe reconnaissance des formes et intelligence artificielle du CRIN ; le développement de recherches sur les bases de connaissances dans cette équipe devrait rapprocher encore nos préoccupations. Merci à tous deux pour leur participation à ce jury.

L'environnement SAIDA n'existerait pas aujourd'hui sans le travail, la compétence et l'enthousiasme de Josette Morinet-Lambert qui en a été la réalisatrice depuis le mois de Septembre dernier. Mais Josette a été aussi pour moi l'interlocutrice exigeante de beaucoup de discussions portant sur le contenu de cette thèse, la compagne des moments difficiles, l'auteur des schémas qui illustrent l'ouvrage et la lectrice attentive des versions finales du document, je lui adresse un grand merci et espère que notre collaboration pourra se poursuivre.

Je remercie aussi tous les collègues du CRIN qui par un renseignement, une référence bibliographique ou un simple mot d'encouragement ont apporté leur pierre à ce document, avec une mention particulière pour François Schwaab qui a continué à partager mon bureau et a su, contrairement à ses habitudes, en être souvent absent et me laisser ainsi le calme nécessaire à l'achèvement de ce travail.

La disponibilité que requiert la rédaction d'une thèse est difficilement compatible avec l'exercice de responsabilités ; j'ai pourtant voulu conserver celles que j'exerçais au ministère de l'éducation nationale et au rectorat de mon académie. Je dois un merci tout particulier au Recteur Claude Mesliand pour la confiance qu'il m'a accordée ; l'intérêt des tâches qu'il m'a confiées, l'enthousiasme avec lequel il exerce ses fonctions et les encouragements qu'il n'a cessé de me prodiguer m'ont souvent donné du coeur à l'ouvrage.

Mais, ces responsabilités, je n'ai pu continuer à les exercer que parce que des collègues ont accepté de me décharger de toutes les tâches pour lesquelles je n'étais pas indispensable; qu'ils sachent que j'ai été consciente de leurs efforts et que j'ai apprécié leur attitude. Denise Kerviel au rectorat et Georges-Louis Baron au ministère ont patiemment filtré les informations tout en me soumettant toujours les problèmes importants, c'était un rôle difficile et je les remercie tout particulièrement de s'en être acquittés avec autant de tact et d'efficacité ; et je n'oublie pas tous les autres que je ne peux citer ici.

Enfin je ne voudrais pas terminer sans mentionner l'appui de Jean-François, Florence, Caroline et Jérôme à qui je réserve bien évidemment d'autres formes de remerciements!

Sommaire

SOMMAIRE

Avant-propos

Avant-propos

Cet ouvrage concrétise des réflexions et travaux menés simultanément ou successivement depuis de nombreuses années dans trois domaines de l'informatique : la construction de programmes, l'intelligence artificielle, notamment ses applications à la conception de bases de connaissances, l'enseignement de l'informatique et l'utilisation de l'informatique dans l'enseignement.

Chronologiquement, c'est en intelligence artificielle que ma réflexion et mon expérience sont les plus anciennes ; c'est en effet au sein de l'équipe de Jacques Pitrat (CNRS et PARIS VI) que j'ai commencé mes recherches [1], [2]. Depuis, je n'ai jamais cessé de travailler dans ce domaine; j'y ai vu émerger les concepts de "système expert" et de "base de connaissances" et je les ai, au fur et à mesure, présentés à des publics variés, notamment à des publics d'enseignants, ce qui m'a rapidement conduite à m'interroger sur les applications possibles de ces techniques dans l'éducation [3], [4].*

Mes auditoires d'enseignants, d'ingénieurs et d'étudiants étaient d'origines et de disciplines différentes; je me suis attachée à leur montrer des applications de l'intelligence artificielle dans des secteurs très divers [5]. Par ailleurs, le CRIN est souvent sollicité pour des collaborations avec le monde industriel ; j'ai ainsi eu l'occasion de participer à la création d'une maquette de système expert d'aide au dépannage d'un haut fourneau avec l'équipe d'ingénieurs d'une société sidérurgique [6], [7].

La nécessité d'organiser ces expériences foisonnantes, l'enseignement que j'ai fait de ces notions en DESS intelligence artificielle, m'ont conduite à m'intéresser aux méthodes nécessaires [8] pour la construction de ces systèmes et expliquent aujourdhui ma participation au PRC-GRECO intelligence artificielle dans le pôle 4 "méthodes et outils" dont la direction est assurée par Jean-Pierre Laurent.

Les numéros renvoient à la page de références relatives à l'avant-propos située en fin de bibliographie.

Par ailleurs, j'ai enseigné la construction d'algorithmes et de programmes à de nombreuses générations d'étudiants et d'adultes. La réflexion sur ce thème a été continue et féconde dans notre laboratoire, sous la direction de Claude Pair puis de Jean-Pierre Finance, aussi bien avec des finalités d'enseignement, que dans le cadre de recherches plus théoriques sur les programmes [9] ou plus appliquées sur la définition de méthodes et la construction d'environnements de programmation [10], [11]. J'ai donc évolué de la syntaxe FORTRAN aux types abstraits de données en passant par la programmation structurée et quelques autres étapes.

Enfin, j'ai, depuis 1974, souvent apporté ma contribution au développement des utilisations pédagogiques de l'informatique, notamment dans l'enseignement secondaire. Cette contribution s'est concrétisée par des formations d'enseignants à l'informatique (IREM, écoles d'été, MAFPEN) et par l'animation de groupes de recherches pédagogiques à l'IREM [12] et au centre de ressources académiques en informatique pédagogique [13]. J'ai aujourdhui la responsabilité de définir et de proposer des orientations pour la politique académique dans ce domaine.

De façon complémentaire, j'ai été amenée à participer à l'introduction de l'informatique comme discipline optionnelle dans l'enseignement général des lycées, d'abord pour former des enseignants et construire avec eux, de façon expérimentale, une pédagogie adaptée à ce public nouveau, puis pour proposer les objectifs, contenus et modalités de mise en oeuvre d'un tel enseignement, comme présidente de Comité Scientifique National chargé du pilotage de cette option auprès du directeur des lycées et collèges [14].

La construction de programmes, l'enseignement de l'informatique et l'informatique dans l'enseignement, les systèmes à base de connaissances sont donc des sujets auxquels j'ai longuement réfléchi avec la participation d'acteurs très divers qui ont tous, chacun à leur manière, contribué à faire avancer cette réflexion.

C'est à la suite de ces expériences variées que j'ai conçu en 1985 le projet de développer un système à base de connaissances sur le choix de représentations de données à des fins pédagogiques. SAIDA a été proposé, en 1986, comme réponse à l'appel d'offre de l'Agence de l'Informatique pour la réalisation de systèmes experts pour l'enseignement. Ce système fait partie des 12 projets retenus à la suite de cet appel d'offre ; sa réalisation bénéficie d'une aide financière du Ministère de l'Industrie gérée au Centre Lorrain d'Enseignement Assisté par Ordinateur (CLEO) par Maryse Quéré.

Introduction

INTRODUCTION

Le niveau croissant de complexité des problèmes à résoudre en informatique d'une part, l'évolution des méthodes, des logiciels et des matériels d'autre part, modifient l'activité de construction de programmes.

Là où il fallait hier écrire quelques centaines ou milliers de lignes de code, il existe aujourd'hui des logiciels conviviaux permettant à l'utilisateur de dialoguer directement avec son micro-ordinateur. Les traditionnels clients de gros programmes (espace, télécommunications, conduite de process, gestion d'entreprise, etc.) exigent des logiciels toujours plus complexes, plus rapides et plus fiables. Pour de nouveaux domaines moins bien formalisés que les précédents, comme l'aide à la décision financière, la conception architecturale, le diagnostic médical, le caractère déclaratif et modulaire des systèmes à bases de connaissances, leurs facultés explicatives, semblent mieux adaptés aux besoins que l'écriture de programmes classiques.

Pour construire ces applications, l'informaticien dispose de plus en plus souvent de stations de travail et développe ses programmes dans un environnement LISP, ADA, ou autre, lui apportant une assistance et un confort de travail grandissants.

Dans ce contexte en évolution toujours rapide, les responsables de formation s'interrogent : que faut-il enseigner aux futurs professionnels de l'informatique dans le domaine de la construction de programmes ? Comment faut-il enseigner ? Peut-on proposer des outils qui facilitent la tâche des élèves et des enseignants ?

Notre travail se situe dans cette perspective ; il est une contribution à la réflexion commune sur l'étude de la programmation et de son enseignement ; il propose, avec les bases de connaissances, une approche originale pour l' enseignement de techniques jusqu'ici assez peu systématisées.

Il faut dans ce vaste thème délimiter un champ d'étude particulier et fixer des objectifs précis. Nous le faisons en mettant en évidence des points importants dans le processus de développement d'une application informatique sur lesquels nous insisterons dans une perspective d'enseignement, et en faisant des choix parmi différentes approches possibles.

Pour maîtriser la complexité des situations, l'homme dispose d'outils intellectuels

puissants nommés abstraction, décomposition en modules, structuration. L'informaticien doit apprendre à s'en servir pour concevoir et réaliser les applications qu'on lui confie. **Nous nous attachons d'abord à l'abstraction et nous souhaitons fournir des modes de description des problèmes et de leurs solutions au niveau d'abstraction requis par ces applications.**

L'informaticien a pour objectif de concevoir et réaliser des logiciels efficaces. Il lui faut donc savoir caractériser les transformations qui, à partir d'une expression abstraite de solution, conduisent à un programme efficace. **Nous nous limitons pour conduire cette réflexion à des solutions exprimées dans des langages impératifs de haut niveau.**

Dans ce cadre, réaliser un programme efficace, c'est, la plupart du temps, choisir des représentations d'informations qui permettent de minimiser le temps ou la place mémoire nécessaires pour l'exécution des opérations effectuées sur ces informations. Il existe pour les structures abstraites d'information les plus courantes (ensembles, listes, piles, tables, etc...) des représentations efficaces et des algorithmes parfois complexes pour optimiser certaines opérations.

Passer d'un algorithme abstrait à un programme efficace, c'est alors le plus souvent faire des choix dans cet outillage de représentations et d'algorithmes disponibles. **Enseigner la construction de programmes, c'est donc notamment donner aux élèves des critères de choix.**

Nous faisons démarrer notre étude avec une étape de travail comportant un algorithme sur des objets abstraits et nous guidons ensuite l'élève dans des choix de représentations qui le conduisent à un programme efficace. L'analyse des manuels et traités de programmation montre que si l'on peut facilement mettre à la disposition de l'élève de nombreux algorithmes implantant des opérations pour des représentations variées des informations, les critères précis permettant de choisir une bonne représentation dans un cas particulier sont rares. En effet, nous touchons là à une phase de l'activité de construction de programmes qui a été très peu formalisée parce qu'elle fait appel à des raisonnements nuancés au cours desquels il faut le plus souvent opérer des compromis entre des exigences a priori contradictoires.

Nos objectifs sont donc d'exprimer cette expertise non dite en matière de représentation d'informations, puis de proposer un moyen de l'enseigner. Les systèmes à base de connaissances fournissent une solution tout à fait adaptée à ces deux objectifs.

Sur le plan de la programmation, nous proposons une base de connaissances permettant de trouver une représentation efficace pour un objet abstrait à partir de caractères incluant des contraintes du problèmes, des propriétés de l'objet à représenter et des opérations

effectuées dans l'algorithme. Il fallait préciser les structures abstraites d'informations envisagées et les opérations effectuées sur ces structures ; cette base de connaissances est donc relative à une bibliothèque de types et d'implantations de types disponible avec le système et programmée en ADA, langage choisi pour l'expression des programmes dans notre réalisation.

Sur le plan de l'enseignement, nous utilisons les possibilités de structurations différentes conduisant à des raisonnements différents et les facultés explicatives des systèmes à base de connaissances. Il s'agit cette fois d'exposer ces raisonnements, de les organiser et de les mettre à la disposition de l' élève de telle façon :

- qu'ils lui soient compréhensibles,
- qu'ils tiennent compte de ce qu'il a appris et de la façon dont il l'a appris,
- qu'ils induisent chez lui une méthode de travail qu'il pourra conserver en l'absence d'assistance du système.

La construction de bases de connaissances et de systèmes à base de connaissances pour des applications pédagogiques sont des activités pour lesquelles on ne possède que peu de méthodes ; nous nous sommes donc fixé comme autre objectif de décrire précisément nos processus de construction afin d'en souligner les étapes importantes et les difficultés de réalisation et de contribuer à la réflexion commune sur ce plan des méthodes.

Notre travail est au carrefour de trois domaines: la construction de programmes, les systèmes à bases de connaissances et l'enseignement. Il a nécessité des études et des réflexions dans les trois domaines qui sont concrétisées par la réalisation d'une maquette d'environnement d'enseignement disponible sur SUN et nommé SAIDA , pour "Système d'Aide à l'Implantation de Données Abstraites".

Ce mémoire est organisé de la façon suivante : un premier chapitre précise les problèmes de construction de programme que nous envisageons, rappelle les difficultés d'enseignement rencontrées et met en évidence le caractère approché des raisonnements à conduire pour effectuer des choix de représentation pour des objets abstraits.

Nous décrivons ensuite l'évolution de la programmation (chapitre 2), les tentatives d'automatisation de la programmation (chapitre 3) et des logiciels utilisés pour l'enseignement de l'informatique (chapitre 4). Ces synthèses dépassent le cadre strict que nous avons fixé à notre étude et ont pour objectif de nous permettre de justifier nos choix et de situer notre contribution dans un contexte plus large.

Une seconde partie est consacrée à l'étude de la matière à enseigner: les types abstraits ou structures abstraites de données (chapitre1), leurs représentations (chapitre2), l'expression des algorithmes les utilisant (chapitre3).

La troisième partie porte sur la conception des composants d'un environnement d'enseignement pour le choix de représentations de données abstraites : une bibliothèque de types et de représentations de types (chapitre 1), une base de connaissances contenant l'expertise en programmation (chapitre 2), la construction d'un système d'enseignement à partir d'une telle base (chapitre 3).

Le système SAIDA est présenté dans la quatrième partie. Dans le premier chapitre est abordé le point de vue fonctionnel, possibilités offertes aux enseignants pour paramétrer l'environnement selon les populations d'étudiants visées, aides mises à la disposition des élèves pour passer d'un texte ADA portant sur des objets abstraits à un programme exécutable efficace. La description de la réalisation, les logiciels utilisés, les difficultés d'intégration rencontrées font l'objet du chapitre 2. Une réflexion plus générale sur la construction d'environnements pédagogiques comportant des bases de connaissances est présentée au chapitre 3.

Enfin, ce projet, conçu à la fois comme réalisation répondant à un besoin identifié et comme champ d'expérimentation d'idées, s'est révélé, jour après jour, source extrêmement féconde de questions nouvelles auxquelles il n'était pas possible de répondre dans le cadre de cette thèse, mais qui sont évoquées tout au long des chapitres et rassemblées dans la conclusion pour ouvrir la voie aux recherches fondamentales et à la construction des outils qui permettront de réaliser mieux et plus facilement d'autres systèmes de ce type.

LE PROBLEME ET SES CONTEXTES

I° Partie

Introduction à la partie I

Notre objectif dans cette première partie est de préciser le problème de construction de programme auquel nous nous intéressons d'un point de vue pédagogique et de décrire des contextes dans lesquels il nous semble important de le replacer, notamment l'évolution de la programmation, les tentatives d'automatisation de cette activité et l'utilisation de l'informatique comme outil pédagogique.

Dans le premier chapitre, nous rappelons à partir d'exemples ce qu'est la programmation par niveaux d'abstractions successifs ; nous montrons comment exprimer un algorithme portant sur des objets abstraits et parvenir à un programme exécutable en ADA, le langage que nous avons retenu pour l'environnement d'enseignement que nous construisons. Nous faisons une première analyse de la nature des raisonnements mis en oeuvre dans le processus de choix d'une implantation d'objet abstrait. Nous justifions ainsi l'approche à base de connaissances retenue dans notre système.

Le second chapitre retrace l'historique de l'évolution de la représentation des informations dans les applications informatiques, avec une mention particulière pour la notion de type dans les langages algorithmiques.

Nous décrivons dans un troisième chapitre différentes tentatives de construction automatique de programmes. Nous y remarquons notamment des réalisations ayant des caractères voisins de celle que nous envisageons. En effet, les projets concernant la construction de programmes réels, que nous opposons ici aux exemples d'école, sont des projets de construction assistée et non d'automatisation totale ; de plus, certains d'entre eux intègrent l'idée de bases de connaissances contenant l'expertise du domaine.

Enfin nous proposons un logiciel d'assistance à l'enseignement ; nous nous intéressons donc de façon plus générale aux différentes fonctions pédagogiques des logiciels et décrivons quelques réalisations dans l'enseignement de la discipline informatique.

Chapitre 1

ECRIRE DES ALGORITHMES ABSTRAITS ET FAIRE DES CHOIX DE REPRESENTATIONS EFFICACES

L'idée que la complexité des applications traitées en informatique rend indispensable l'écriture d'algorithmes à un niveau proche des problèmes est communément admise [DAH,72], [BOO,83], [LIS,86]. Plusieurs cadres s'offrent aujourd'hui au concepteur d'applications pour modéliser son univers de travail et les traitements à effectuer sur les informations de cet univers. A coté de langages dotés de primitives de haut niveau comme SETL, CLU, ADA, des environnements orientés objets comme SMALLTALK ou plus récemment EIFFEL, la logique avec PROLOG ou le modèle relationnel des bases de données sont de plus en plus utilisés.

Mais il est également bien connu qu'un principe comme celui que nous venons d'énoncer ne peut avoir de réalité dans le quotidien du programmeur que s'il est accompagné d'outils efficaces permettant de le mettre en œuvre aisément. Abstraction dans l'expression de l'algorithme ne doit pas s'opposer à efficacité dans le programme produit. Il est donc nécessaire d'offrir à la fois une aide à la conception des algorithmes à un niveau abstrait et une assistance au choix d'implantations efficaces de données.

Avant tout développement, nous voulons préciser davantage dans ce chapitre le cadre et les objectifs de notre étude.

Nous présentons, à travers plusieurs exemples, la famille de problèmes à laquelle nous nous intéressons. Pour la résolution de ces problèmes, nous nous limitons à l'utilisation de langages impératifs dits de haut niveau et nous expliquons ce que nous entendons par algorithme portant sur des données abstraites, par critère de choix d'implantations efficaces, par environnement permettant à la fois d'exprimer un algorithme sur des types de données adaptés au problème à résoudre et de représenter les objets de cet algorithme dans le langage de

programmation cible.

Nous développons complètement un petit problème de recherche de mots-clés dans un texte qui nous servira à plusieurs reprises de référence dans les chapitres suivants ; nous analysons ensuite deux applications moins triviales afin d'illustrer la diversité, la complexité des situations rencontrées, les nuances à apporter dans la proposition de représentations efficaces d'informations abstraites avec les constructions syntaxiques des langages usuels.

Nous mettons ainsi en évidence la nécessité d'expliciter les raisonnements du programmeur dans cette phase de travail. Cette "expertise" peut alors être utilisée pour automatiser, au moins partiellement, le passage d'un algorithme abstrait à un programme efficace et pourrait servir de base à un compilateur de langage abstrait. Ce n'est pas notre objectif prioritaire dans ce travail ; nous nous intéressons surtout à la façon de faire acquérir cette expertise par des élèves. Les exemples et propositions de ce chapitre sont donc placés dans une perspective pédagogique.

1.1 PROGRAMMER AVEC DES TYPES ABSTRAITS DE DONNEES

1.1.1 Un premier exemple

Lorsqu'on veut faire expérimenter par des élèves la notion de type de donnée, on leur fait analyser des énoncés comme le suivant :

" On donne un texte et des mots-clés ; écrire un algorithme qui permet de compter les mots du texte qui sont des mots-clés ; un mot figurant plusieurs fois dans le texte sera compté autant de fois qu'il apparaît. "

Pour écrire une première version d'algorithme résolvant ce problème, il faut :

- représenter les objets de l'univers du problème dans un modèle qui, parmi ceux proposés, paraît bien adapté,
- concevoir et exprimer une suite de calculs sur ces objets qui conduise au résultat cherché.

Pour illustrer l'utilisation du type ensemble muni de l'opération "appartient" et du type "liste" muni d'une opération de parcours, on peut proposer la solution suivante :

Le texte est représenté par la liste des mots qui le composent et les mots-clés par un ensemble, cette idée conduit à l'ébauche d'algorithme suivante :

Pour chaque mot du texte **faire**
    si ce mot appartient à l'ensemble des mots-clés **alors** le compter **finsi**
finpour.

Cette version d'algorithme est exprimée à l'aide d'un pseudo-langage, à l'image de ceux que les informaticiens emploient souvent. Elle utilise des opérations comme le parcours d'une liste et le test d'appartenance d'un élément à un ensemble. Si nous disposons d'un langage nous permettant de nous exprimer d'une façon analogue, si dans cet environnement, il existe des types d'objets et des opérateurs de profils bien définis pour traduire les opérations dont nous avons besoin, nous pouvons écrire notre première version d'algorithme dans cet environnement. Si de plus, nous disposons ensuite d'une bibliothèque de représentations variées pour chacun de ces types, il suffit de sélectionner la représentation qui paraît la plus souhaitable pour chaque objet et d'ajouter les modules correspondants à l'algorithme abstrait pour obtenir un programme exécutable.

Une longue expérience d'enseignement nous a montré qu'il était difficile d'obtenir une expression d'algorithme au niveau des problèmes lorsque cette expression ne peut se faire que sur le papier ; la tentation de passer directement au langage de programmation est trop forte et il n'y a pas de réflexion sur d'éventuels critères de choix de représentation de données. Il paraît donc souhaitable de faire travailler les étudiants dans un environnement où la version abstraite de l'algorithme n'est pas une étape artificielle, mais est partie intégrante du programme final.

Pour illustrer cette notion d'environnement de construction de programmes, nous choisissons, parmi plusieurs langages qui présentent ces caractéristiques, d'exprimer nos solutions en ADA. Ce choix nous semble justifié notamment par la diffusion actuelle de ce langage dans l'industrie du logiciel et par les besoins de formation induits par cette diffusion. Mais, la plupart des idées que nous avançons par la suite sont largement indépendantes de ce choix de langage d'expression.

Nous développons donc maintenant une solution en ADA, en présentant successivement les spécifications de types abstraits utilisables, l'algorithme résolvant le problème et la façon de sélectionner l'une des représentations disponibles pour chaque objet déclaré.

1.1.2 Une expression de la solution en ADA

Nous avons utilisé les types liste et ensemble pour exprimer notre algorithme ; pour donner à ces structures un degré de généralité suffisant, il est nécessaire de pouvoir les paramétrer par le type des éléments qu'elles contiennent et par des noms de fonctions que les

algorithmes implantant les opérateurs peuvent être amenés à utiliser. Les notions de paquetage et de généricité en ADA sont une réponse possible à ces exigences.

La figure 1 montre un extrait de la spécification utilisée pour le type générique LISTE. On peut y observer :

- que ce paquetage est paramétré par ELEM qui représente le type des éléments de la liste,
- qu'il offre notamment sur les objets de type LISTE une procédure d'initialisation CREER_LIST paramétrée par le nom de la liste et une procédure PARC_TOTAL qui est un parcours de la liste paramétrée par la procédure FONC ; FONC applique à chaque élément de la liste le traitement défini dans son corps de procédure.

La figure 2 montre des extraits de la spécification d'un paquetage pour un type générique ENSEMBLE. On y trouve une fonction APPARTIENT paramétrée par le nom de l'ensemble et un nom d'élément que nous pourrons utiliser pour la procédure COMPTE_MOTS.

Dans cet environnement, la procédure ADA pour compter les mots-clés s'écrit comme le montre la figure 3. On y lit la déclaration du type "CHAINE" des éléments manipulés, ici des chaînes de caractères, l'instanciation d'un type MA_LISTE à partir de la spécification fournie, l'instanciation d'un type MON_ENSEMBLE. On définit ensuite l'action à effectuer sur chaque élément E de la liste dans la procédure INCREMENTER et on crée un exemplaire de la procédure générique PARC_TOTAL qui est nommé PARCOURS et paramétré par INCREMENTER. Une fois ces "déclarations" achevées, l'algorithme abstrait tient entre "begin" et "end" ; il est complété pour plus de réalisme par des procédures d'initialisation des objets par lecture.


```
-- ------------------------------------------------------------------------------ --
--                        fichier : lis_manager_chn_spec.a                        --
--      définition d'une interface pour la manipulation des objets du type abstrait "liste"  --
--                        implantation par liste chaînée simple                   --
-- ------------------------------------------------------------------------------ --
with SYSTEM ;
generic
        type   ELEM is private ;

package LIS_MANAGER_CHN is

        type LISTE is private ;
        subtype TRANG is INTEGER range 0 .. SYSTEM.MAX_INT ;
        type PLACE is private ;

        procedure CREER_LIST ( IDLIST : in out LISTE ) ;
        -- création d'une liste vide en positionnant sa taille, sa tête et sa queue

        procedure ADJL( IDLIST: in out LISTE; RANG: in TRANG ; IDVAL : in ELEM ) ;
        -- adjonction d'un élément IDVAL qui occupe le rang indiqué dans la liste IDLIST
        :
        :

        generic
                with procedure FONC( E : in out ELEM ) ;
        procedure PARC_TOTAL( IDLIST : in out LISTE ; PL : out PLACE ) ;
        -- permet d'appliquer la procédure FONC à chacun des éléments de la liste IDLIST,
        -- retourne une place, celle du dernier élément si tout s'est bien passé,
        -- celle où il y a eu un problème sinon
        :
        :
        private

        type   ELEMENT ;
        type   PLACE is access ELEMENT ;
        type   ELEMENT is
               record
                      VAL       : ELEM ;
                      SUIVANT   : PLACE ;
               end record ;

        type   LISTE is
               record
                      T         : TRANG ;
                      TETE      : PLACE ;
                      QUEUE     : PLACE ;
               end record ;
        end LIS_MANAGER_CHN ;
```

Figure 1 : extraits de la spécification d'un paquetage de type LISTE.

```
-- ------------------------------------------------------------------------------------------ --
--                                  fichier : ens_manager_tbv_spec.a                          --
--     définition d'une interface pour manipuler des objets de type abstrait "ensemble"       --
--                          implantation d'ensemble par tableau de valeurs                    --
--                        dont la taille est limitée lors de la déclaration                   --
-- ------------------------------------------------------------------------------------------ --
:
:
generic
        type ELEM is private ;
        :
        :
package ENS_MANAGER_TBV is
        Subtype TRANG is Natural range 0.. SYSTEM.MAX_INT ;
        type ENS (T_MAX : TRANG) is private ;

        procedure CREER( E :  out ENS  ) ;
        procedure EST_VIDE( E : in ENS ; OK :  BOOLEAN ) ;
        procedure AJOUTER( E : in out ENS ; X : in ELEM ) ;
        procedure APPARTIENT( E : in ENS ;  X : in ELEM ; OK : BOOLEAN ) ;
        procedure UNION( E1, E2 : in ENS ; E3 : in out ENS) ;
        procedure INTER( E1, E2 : in ENS ; E3 : in out ENS) ;
        procedure CARDINAL( E : in ENS ; N :  INTEGER ) ;
        :
        :

        private
        type TABV is array (Natural range <>) of ELEM ;
        type ENS (T_MAX : TRANG) is
                record
                        TAILLE : TRANG ;
                        TVAL : TABV(0..T_MAX) ;
                end record ;


        end ENS_MANAGER_TBV ;
```

Figure 2 : Extraits de la spécification d'un paquetage de type ENSEMBLE.


```
with LIS_MANAGER_CHN, ENS_MANAGER_TBV, TEXT_IO ; use TEXT_IO ;
procedure COMPTE_MOTS is
        subtype CHAINE is STRING ( 1 .. 10 ) ;
        package MA_LISTE is new LIS_MANAGER_CHN (ELEM => CHAINE ) ;
        use MA_LISTE ;
        -- cette ligne instancie le type liste de mots à partir du type générique LIS_MANAGER_CHN
        -- disponible dans notre environnement
        TEXTE : LISTE ;
        PLACE_TEXTE : PLACE ;
        COMPTE : INTEGER ;

        package MON_ENSEMBLE is new ENS_MANAGER_TBV (ELEM => CHAINE ) ;
        use MON_ENSEMBLE ;
        -- cette ligne instancie le type ensemble de mots à l'aide du type générique
        -- ENS_MANAGER_TBV disponible dans  notre environnement
        MOTS_CLES : ENS (20) ; -- contrainte de taille à fixer pour ENS

        package MON_ENTIER is new INTEGER_IO ( INTEGER ) ; use MON_ENTIER ;

        procedure COMPTER (L : in out LISTE ; ENS :  in  ENS ; COMPTEUR : in out INTEGER ) is
                PLACE : TRANG ;
                procedure INCREMENTER ( E : in out CHAINE ) is
                        EST_DANS : BOOLEAN ;
                        begin
                                APPARTIENT ( ENS , E, EST_DANS ) ;
                                if  EST_DANS then
                                        COMPTEUR := COMPTEUR + 1 ;
                                end if ;
                        end INCREMENTER ;
                procedure PARCOURS is new PARC_TOTAL ( FONC => INCREMENTER ) ;
                -- instancie la procédure parcours à partir de la procédure générique
                -- PARC_TOTAL de l'environnement en définissant la procédure incrémenter

                begin
                        COMPTEUR := 0 ;
                        PARCOURS ( L , PLACE ) ;
                end COMPTER ;
        procedure INIT_PAR_LECTURE ( IDLIST : in out LISTE ) is
                begin
                ....
                end INIT_PAR_LECTURE ;
        procedure INIT_PAR_LECTURE ( IDSET : in out ENS ) is
                begin
                ....
                end INIT_PAR_LECTURE ;

        begin
                CREER_LIST ( TEXTE ) ;
                INIT_PAR_LECTURE ( TEXTE ) ;
                CREER ( MOTS_CLES ) ;
                INIT_PAR_LECTURE ( MOTS_CLES ) ;
                COMPTER  ( TEXTE, MOTS_CLES, COMPTE ) ;
                PUT ( COMPTE ) ;
        end COMPTE_MOTS ;
```

Figure 3 : Procédure compte-mots.

Nous venons de montrer à travers cet exemple comment passer d'un algorithme abstrait à une procédure ADA en instanciant correctement les types et procédures fournis par une bibliothèque prédéfinie.

A ce stade de réalisation, un choix implicite de représentation des listes et ensembles a été fait lorsque des noms de paquetages de l'environnement ADA ont été cités pour instancier les types MON_ENSEMBLE et MA_LISTE. Changer de représentation revient donc, dans les cas simples, à changer les noms de paquetage dans le texte de la figure 3, c'est une opération facile à réaliser lors de la création d'un programme sous éditeur ; nous pouvons donc nous attacher maintenant à préciser comment analyser cette procédure et son contexte pour faire un choix raisonné d'implantation de l'ensemble et de la liste utilisés.

Cette étude rapide peut avoir laissé des interrogations chez le lecteur non familier de ADA. Ce dernier trouvera une présentation systématique des concepts du langage ADA que nous utilisons dans la seconde partie de l'ouvrage (chapitre 3).

1.2 DES CHOIX DE REPRESENTATION DES DONNEES.

1.2.1 Analyse d'un exemple (suite)

Le premier objet déclaré dans la procédure de la figure 3 est TEXTE de type liste de chaînes de caractères, la seule opération effectuée sur cet objet, mis à part l'algorithme de création, est un PARCOURS séquentiel appliquant à chaque élément du texte la procédure INCREMENTER ; beaucoup de représentations de liste permettent cette opération et son efficacité varie peu entre une représentation dans un tableau de chaînes de caractères et une représentation chaînée avec pointeurs. Cette opération ne fournit donc pas ici un critère prépondérant pour le choix d'une représentation.

Par contre d'autres informations sont nécessaires, qui ne figurent pas dans l'algorithme. Quelle est la longueur du texte ? Est-il en mémoire centrale au moment de l'exécution de la procédure ou son maintien sur fichier externe est-il imposé ? Y-a-t-il d'autres données, de longueur variable selon les exécutions, en même temps que le texte ?

La seconde donnée déclarée est MOT_CLES de type ensemble de chaînes de caractères. La seule opération effectuée fréquemment dessus est l'application de la procédure APPARTIENT qui effectue le test d'appartenance d'un mot du texte à l'ensemble des mots-clés. En l'absence d'autres indications ou contraintes, c'est donc une représentation optimisant l'exécution de cette opération qu'il faut choisir. Parmi les représentations possibles pour les ensembles, toutes ne sont pas équivalentes pour cette opération : une représentation

par tableau de booléens indexé par le type des éléments de l'ensemble est, par exemple, particulièrement efficace ; mais ici le type chaîne de caractères ne nous permet pas de choisir cette représentation. On peut alors conseiller de ranger les mots-clés dans un tableau de mots trié, puisqu'il n'y a ni adjonction ni suppression dans cet ensemble, et d'y implanter le test d'appartenance par une recherche dichotomique.

Retenons de cet exemple simple que le choix d'une représentation fait appel à des caractéristiques variées de l'objet à implanter (sa taille), de l'environnement (fichier externe imposé ou non) et des opérations utilisées dans l'algorithme (parcours, test d'appartenance). Ce sont donc les relations entre des caractéristiques de ce type et les représentations disponibles que nous devons nous attacher à expliciter ; nous le faisons dans la suite de ce chapitre en analysant deux problèmes plus conséquents.

1.2.2 Le hit-parade musical

Enoncé

Il s'agit de dépouiller une enquête relative aux chansons les plus populaires. Les personnes interrogées ont fourni leur nom, leur sexe, leur âge et 5 numéros de chansons. On veut obtenir :

- un classement des chansons avec, pour chacune d'elles, le nombre de fois où elle a été mentionnée.
- une liste des personnes ayant cité un des trois premiers scores dans chacune des catégories suivantes : sexe masculin et plus ou moins 20 ans, sexe féminin et plus ou moins de 20 ans.

On traite de l'ordre de 5000 réponses portant sur 100 chansons.

Intérêt du problème

Dans une perspective pédagogique, il faut distinguer une phase de recensement et de modélisation des objets de l'univers, et nous nous limitons ici à l'utilisation de types de base et de constructeurs de types usuels (listes, tables, ensembles), puis une phase de choix de représentation pour chacun des objets. Nous ne discutons pas des critères de choix qui pourraient intervenir à la première étape, nous donnons seulement le résultat de cette première étape qui constitue notre point de départ.

Il faut également remarquer que nous envisageons ici un exemple demandant de représenter non seulement les informations décrites dans l'énoncé mais aussi des intermédiaires servant à la construction de la solution, comme les scores associés aux chansons permettant d'obtenir le classement des chansons. Par ailleurs, les quatre listes correspondant aux quatre catégories de personnes indiquées dans l'énoncé ne sont pas indépendantes sur le plan de la place à prévoir en mémoire pour leur représentation ; on ne peut prévoir la longueur moyenne de chacune d'elles, mais la somme de leurs nombres d'éléments est certainement

inférieure au nombre total de réponses.

Objets à représenter

Les données : Les 5000 réponses sont exploitées les unes après les autres et peuvent être modélisées par une liste.

Les résultats : Le classement des chansons identifiées par leur numéro associé peut être représenté par une liste de couples (chanson, score) triée par numéros de score décroissants.

Les personnes, par catégorie, ayant fournies l'une des trois meilleures réponses, sont représentées par des ensembles.

Les informations intermédiaires : La liste de couples (chanson, score) peut résulter d'une table (numéro chanson -> score) dans laquelle on effectue un changement de valeur de score chaque fois qu'une chanson est citée.

L'algorithme est simple et il n'y a pas d'autre structure de données intermédiaire à introduire que la table permettant de totaliser les scores par chanson. Il s'agit donc maintenant d'analyser les caractéristiques du problème qui vont servir à choisir des implantations pour les objets utilisés.

Choix d'implantation

La liste des réponses est utilisée avec une opération de parcours, sa taille est fixe ; elle peut donc être représentée par un fichier en mémoire secondaire ou un tableau en mémoire centrale.

La table servant au calcul des scores a un nombre d'entrées connu et relativement petit; elle peut être implantée par un tableau indicé sur l'ensemble des numéros de chansons, c'est à la fois la représentation la plus économique en place mémoire et la plus rapide en temps d'accès à une information.

La liste indiquant le classement des chansons et leur score ne supporte ni adjonction ni suppression, sa taille est connue, une représentation contiguë dans un tableau de 100 éléments convient, chaque élément du tableau étant un couple (chanson, score).

Restent à implanter les ensembles par catégories ; on ne connait pas a priori la taille de ces objets ; on sait seulement que dans les plus mauvais cas chacun peut atteindre les 5000 éléments ; on sait également que la somme des éléments contenus dans les quatre ne peut dépasser 5000. Il n'y a pas de problème de rapidité d'accès à une information puisqu'il ne

s'agit que de création par adjonctions successives et sans ordre particulier. Une implantation en listes chaînées optimisera l'espace mémoire utilisé et évitera 4 tableaux dimensionnés chacun à 5000 éléments. Remarquons que nous utilisons ici le terme "liste chaînée" par abus de langage pour décrire une forme de structuration de données dans un langage permettant de représenter les éléments d'un ensemble.

1.2.3 Le problème de la session d'examen

Nous empruntons cet exemple à Hoare dans ses "Notes on Data Structuring" [DAH,72] ; il a le double avantage de proposer un thème de travail a priori familier à des étudiants et de mettre en jeu une dizaine d'objets complexes pour chacun desquels un choix de représentation est à faire.

Enoncé

Le problème envisagé ici est celui de l'organisation d'une session d'examen dans une université. L'enseignement est dispensé en modules ; chaque étudiant suit plusieurs modules et l'organisation de la session doit obéir à certaines contraintes :

- tout étudiant doit pouvoir se présenter aux épreuves des modules auxquels il est inscrit (avec une limite supérieure du nombre de modules par étudiant).
- la salle d'examen ne peut contenir plus d'un certain nombre d'étudiants.
- l'ensemble de la session doit être aussi court que possible (mais on ne se pose pas de problème mathématique d'optimisation).
- l'emploi du temps sera organisé en demi journées, chaque étudiant n'ayant qu'une épreuve à passer par demi journée.

Les informations disponibles pour résoudre se problème sont les suivantes :

- épreuves auxquels chaque étudiant est inscrit,
- nombre total d'épreuves,
- nombre maximum d'épreuves pour une demi journée,
- nombre maximum d'étudiants pouvant composer ensemble,
- nombre maximum d'épreuves par étudiant.

Formalisation de l'Univers

Un étudiant et une épreuve sont des objets non structurés, nous supposons que nous disposons des types énumérés correspondants. Les informations dont on a effectivement besoin pour construire l'emploi du temps de la session sont en fait les suivantes :

- le nombre de candidats inscrits à chaque épreuve,
- les épreuves qui ne peuvent avoir lieu en même temps qu'une épreuve

donnée.

Ces deux informations, auxquelles on aura besoin d'accéder souvent à partir d'une épreuve donnée, peuvent être modélisées par des tables.

NBCANDIDATS : EPREUVES → 0...NMAX

où NMAX est le nombre maximum de candidats possible pour une épreuve.

INCOMPATIBLES : EPREUVES → ENS de EPREUVES

Ces deux tables peuvent être construites à partir d'un simple parcours des données qui renferment, rappelons-le, pour chaque étudiant la liste des épreuves auxquelles il est inscrit. Cette liste est modélisée par une LISTE, nommée L_ETUDIANTS, de ENS de EPREUVES.

En utilisant les notations de Hoare, les opérations union et différence entre deux ensembles, la phase d'initialisation des deux tables NBCANDIDATS et INCOMPATIBLES s'écrit donc :

table NBCANDIDATS = Ø

table INCOMPATIBLES = Ø
pour chaque ET **dans** L_ETUDIANTS **faire**
  pour chaque EXAM **dans** ET **faire**
    NBCANDIDATS (EXAM) : +1
    INCOMPATIBLES (EXAM) : U {ET - {EXAM}}
  fpour
fpour

L'algorithme de confection d'emploi du temps va donc travailler avec comme données les tables NBCANDIDATS et INCOMPATIBLES ; il doit fournir comme résultat EMPLOI_DU TEMPS qui est un ensemble d'objets du type "DEMI_JOURNEES", c'est à dire ensemble d'ensembles d'épreuves.

La figure suivante récapitule les types et objets structurés introduits jusqu'à maintenant.

TYPES :	EPREUVES :	énuméré
	ENS_EPREUVES :	ENS de EPREUVES
OBJETS :	L_ETUDIANTS :	LISTE de (ENS_ EPREUVES)
	EMPLOI_DU_TEMPS :	ENS de (ENS_EPREUVES)
	NBCANDIDATS :	TABLE (EPREUVES → 0..NMAX)
	INCOMPATIBLES :	TABLE (EPREUVES → (ENS_EPREUVES))

Algorithme de fabrication de l'emploi du temps

Les choix qui ont conduit aux schémas d'algorithme qui suivent sont expliqués en détail dans [DAH,72] et nous y renvoyons le lecteur intéressé. Nous n'avons retenu que les principales étapes et ne commentons que celles qui font intervenir des objets complexes intermédiaires que nous aurons à représenter au même titre que les objets "données" et les objets "résultats".

Nous allons remplir l'emploi du temps par adjonctions successives d'épreuves dans les demi-journées et de demi-journées dans l'emploi du temps.

Première version :

emploi du temps = Ø
tant que emploi du temps ne satisfait pas
  " chaque épreuve est programmée en respectant les souhaits fixés "
  faire
  sélectionner une demi-journée satisfaisant les contraintes
  ajouter cette demi-journée à l'emploi du temps
fin tant que

La sélection d'une nouvelle épreuve ne peut se faire que parmi celles qui ne sont pas encore programmées, ce qui amène à introduire un objet intermédiaire, les épreuves non encore affectées et à le représenter comme un ensemble

RESTE_A_AFFECTER : ENS de EPREUVES

Le choix des épreuves à placer dans une nouvelle demi-jounée amène à engendrer des ensembles d'épreuves issues de RESTE_A_AFFECTER et à sélectionner celui qui remplit au mieux la salle d'examen, l'idée étant qu'on a ainsi satisfait le plus grand nombre d'étudiants.

L'action "sélectionner une demi-journée satisfaisant les contraintes" peut alors s'écrire:

fonction SELECTION_DEMI_JOURNEE : ENS_EPREUVES ;
  ESSAI, MEILLEUR_JUSQUE_LA, NON_ESSAYE : ENS_EPREUVES ;
  E : EPREUVES ;
  début
  E := CHOISI_DANS (RESTE_A_AFFECTER)
  ESSAI := {E}
  MEILLEUR_JUSQUE_LA := {E}
  NON_ESSAYE := RESTE_A_AFFECTER - ESSAI - INCOMPATIBLES (E)
  GENERER_ENSEMBLES
  SELECTION_DEMI_JOURNEE := MEILLEUR_JUSQUE_LA
  fin SELECTION_DEMI_JOURNEE

algorithme final

D'où finalement l'algorithme complet de la figure suivante :

```
objets de l'univers :

type    ENS_EPREUVES = ENSEMBLE de EPREUVES
        L_ETUDIANTS = LISTE de ENS_EPREUVES  (donnée)
        EMPLOI_DU_TEMPS = ENSEMBLE de ENS_EPREUVES
        NB_CANDIDATS = TABLE (EPREUVES -> 0..NMAX)
        INCOMPATIBLES = TABLE (EPREUVES -> ENS_EPREUVES)

fonctions

fonction NB_ETUDIANTS (ENS_EP : ENS_EPREUVES) : ENTIER
calcule le nombre d'étudiants devant composer pour les épreuves contenues dans ENS_EP
        SOMME := 0
        début
        SOMME := 0
        pour chaque E dans ENS_EP faire SOMME := SOMME + 1 finpour
        NB_ETUDIANTS := SOMME
        fin NB_ETUDIANTS

objet intermédiaire global introduit par les fonctions suivantes :
RESTE_A_AFFECTER : ENS_EPREUVES

fonction SELECTION_DEMI_JOURNEE : ENS_EPREUVES
        E : EPREUVES
        MEILLEUR_JUSQUE_LA, ESSAI, NON_ESSAYE : ENS_EPREUVES
        début
        E := CHOISI_DANS (RESTE_A_AFFECTER)
        MEILLEUR_JUSQUE_LA := {E}
        ESSAI := {E}
        NON_ESSAYE := RESTE_A_AFFECTER - ESSAI - INCOMPATIBLES (E)
        GENER_ENSEMBLES
        SELECTION_DEMI_JOURNEE := MEILLEUR_JUSQUE_LA
        fin  SELECTION_DEMI_JOURNEE

procédure GENER_ENSEMBLES
engendre des suites d'ensembles candidates à former des demi-journées pour la session
        E : EPREUVES
        ENS1, ENS2 : ENS_EPREUVES
        début
        ACTUALISER_MEILLEUR
        ENS1 := NON_ESSAYE
        si TAILLE (ESSAI) < NBMAXEPREUVES alors
            tant que NON_ESSAYE ≠ Ø  faire
                E := CHOISI_DANS (NON_ESSAYE)
                ENS2 := NON_ESSAYE ∩ INCOMPATIBLES (E)
                NON_ESSAYE  : - ENS2
                si NB_ETUDIANTS (ESSAI) < TAILLE_SALLE
                alors GENER_ENSEMBLES
                finsi
                NON_ESSAYE : U ENS2
            fintant que
        ESSAI : -{E}
        finsi
fin  GENER_ENSEMBLES

procédure ACTUALISER_MEILLEUR
```



```
        début
        si NB_CANDIDATS(MEILLEUR_JUSQUE_LA) < NBCANDIDATS(ESSAI)
        alors MEILLEUR_JUSQUE_LA := ESSAI
        finsi
fin  ACTUALISER_MEILLEUR

procédure principale
        S : ENS_EPREUVES
        pour chaque ET dans L_ETUDIANTS faire
                pour chaque EXAM dans ET faire
                        NBCANDIDATS(EXAM) :+1
                        INCOMPATIBLES(EXAM) : U {ET - {EXAM}}
                finpour
        finpour
        tant que RESTE_A_AFFECTER ≠ Ø faire
                début
                S := SELECTION_DEMI_JOURNEE
                EMPLOI_DU_TEMPS : U S
                RESTE_A_AFFECTER : - S
                fin
        imprimer EMPLOI_DU_TEMPS
fin procédure principale
```

choix d'implantations

Pour choisir des implantations d'objets, il faut connaitre quelques caractéristiques complémentaires sur les tailles respectives des objets. On suppose donc remplies les conditions suivantes :

- il n'y a pas plus de 500 épreuves, chacune étant passée par moins de 1000 étudiants, (50 en moyenne),
- il y a 5000 étudiants,
- la salle d'examen contient 1000 places,
- il y a au maximum 30 épreuves par demi-journée (en moyenne 10),
- on ne dépasse pas en général 50 demi-journées pour l'organisation de la session.

. Objets modélisant les demi-journées (ensemble d'examens)

On peut hésiter entre un tableau booléen (500 bits) ou une représentation des éléments effectivement présents (tableau de 30 mots); le gain au niveau espace mémoire est peu significatif. Il faut donc regarder quelles opérations sont faites sur un tel objet. Les principales sont l'adjonction d'un examen n'y figurant pas encore ou le retrait du dernier examen entré. Un tableau des éléments est donc une bonne représentation pour cet ensemble, avec une gestion des adjonctions et suppressions sous forme de pile. On ajoutera au tableau le nombre d'étudiants qui se présentent à cet ensemble d'épreuves et on le tiendra à jour au cours des opérations d'adjonction et de suppression.

. L'emploi du temps

L'emploi du temps est un ensemble de demi-journées obtenu par adjonctions successives ; les

demi-journées n'ont pas le même nombre d'épreuves ; on peut proposer une liste de listes dont on connait le nombre total d'éléments (le nombre d'épreuves puisque chacune doit figurer une fois et une seule). On peut aussi se contenter de noter pour chaque épreuve dans quelle demi-journée elle est affectée, il faut pour cela un tableau ayant autant d'éléments qu'il y a d'épreuves.

. Les épreuves incompatibles, les essais

On applique des opérations ensemblistes à ces ensembles d'épreuves, aussi la meilleure représentation pour ces objets est le tableau de booléens.

Que retenir de cet exemple ? La taille des objets joue un rôle important dans les choix de représentation, la variation de cette taille au cours de l'exécution d'un programme aussi ; nous utilisons là un mélange d'analyse statique d'algorithme et d'analyse dynamique. Des représentations très adaptées aux ensembles, comme les tableaux de booléens, ou induites par des opérations particulières (premier enlevé = dernier entré) comme un tableau géré en pile sont proposées. Elles donnent une idée de la variété des solutions possibles au problème des représentations de données abstraites.

1.3 CONSTRUIRE DES PROGRAMMES AVEC SAIDA

Nous venons de montrer comment exprimer des algorithmes abstraits dans un environnement offrant une bibliothèque de types de données et d'implantations variées pour ces types. Nous avons ensuite mis en évidence quelques uns des critères qui peuvent guider un choix de représentation efficace en décrivant les raisonnements que nous faisons sur les algorithmes et les objets à représenter pour parvenir à ces choix.

Quel cahier des charges proposer à l'issue de cette analyse pour un système d'enseignement en "programmation avancée" ?

Nous avons souvent distingué deux phases dans le processus de construction d'un programme. A l'issue d'une première phase, on dispose d'un algorithme décrivant des traitements sur des objets modélisés à l'aide d'un petit nombre de types et d'opérations sur ces types. On cherche ensuite à choisir, pour chaque objet, une représentation efficace ; et nous avons essayé de mettre en évidence quelques-uns des critères de choix utilisables pour mener à bien cette seconde étape dans un processus de construction de programme.

Un travail qui dépasse de loin les objectifs de cette étude consisterait à partir de types très généraux inspirés par le problème à résoudre et à trouver des critères et des transformations permettant de se ramener aux constructeurs de types que nous avons utilisés

dans les exemples de ce chapitre. Les études actuelles [MEY,87a0] sur les avantages respectifs du typage et de l'héritage dans les langages inciteraient même à savoir choisir entre l'un et l'autre mécanisme. On est encore loin aujourd'hui d'une expertise formalisable dans ce domaine.

Nous proposons de fixer comme objectif à notre application d'aider l'élève dans la seconde phase que nous identifions dans la construction d'un programme. Le système proposé doit donc permettre à l'élève :

- d'exprimer des algorithmes abstraits à l'aide d'une bibliothèque de types et d'opérateurs prédéfinie, pour des problèmes de la famille de ceux que nous venons d'examiner,
- de disposer de techniques de représentations efficaces pour les objets de ces algorithmes,
- de transformer les algorithmes en programmes en choisissant des modules appropriés de l'environnement.

Nous ne proposons pas de méthode de travail ; mais nous espérons que la structuration du raisonnement proposé par le système induira des habitudes d'organisation du travail chez l'élève.

Nous avons vu que faire des choix de représentation, c'est raisonner de façon nuancée et fine sur des caractéristiques d'un algorithme et de son environnement ; nous voulons mettre ces raisonnements à la disposition de l'élève et le guider dans leur conduite :

- en construisant un système à base de connaissances contenant l'expertise du programmeur dans ce domaine,
- en proposant des utilisations différentes de cette base de connaissances selon le projet pédagogique de l'enseignant.

La construction d'un tel système demande à être resituée dans plusieurs cadres. D'une part, sur le plan de la construction des programmes, il faut bien mesurer la place de cette forme de travail dans l'évolution générale des méthodes et outils de la programmation et par rapport aux différentes tentatives d'automatisation du processus de construction de programmes. C'est l'objet des chapitres 2 et 3 qui suivent.

D'autre part, du point de vue de l'enseignement, il faut situer ce système par rapport aux différentes formes d'environnements informatiques à vocation pédagogique, notamment par rapport à d'autres systèmes ayant pour objectif de contribuer à l'enseignement de la programmation. C'est l'objet du quatrième chapitre de cette première partie.

Nous espérons avoir ainsi donné une vision plus concrète des problèmes que nous voulions résoudre à l'aide de notre système et une idée au moins intuitive de nos choix de réalisation.

Chapitre 2

AXES ET REPERES DANS L'EVOLUTION DE L'ACTIVITE DE CONSTRUCTION DE PROGRAMMES

Nous avons présenté au chapitre précédent quelques aspects du processus de construction des programmes et indiqué comment nous proposions de contribuer à améliorer la formation des informaticiens dans ce domaine. Définir des objectifs, des contenus, des méthodes pour former des professionnels de l'informatique impose au préalable de s'interroger sur la nature de l'activité de construction de programmes dans le monde de la production de logiciel, sur ce qu'elle a été dans le passé et sur ce que l'on peut en prévoir pour le futur.

Il nous paraît en effet important pour l'enseignant - et probablement aussi pour l'élève - non seulement, de maîtriser les savoirs et savoir-faire à transmettre à un moment donné, mais aussi de comprendre comment cet état de l'art s'est peu à peu construit et a émergé comme contenu scientifique autonome et identifiable à partir de tentatives diverses. C'est probablement la meilleure préparation pour permettre d'anticiper quelque peu les évolutions à venir.

L'informatique est une science jeune, elle a une histoire déjà fort riche sur le plan des concepts mais qui nous est facilement accessible, il faut en profiter. L'objectif de ce chapitre est donc double, d'une part fournir une photographie de ce qu'est la construction de programmes aujourd'hui et anticiper ce qu'elle sera demain en analysant les orientations et productions de la recherche dans ce domaine, d'autre part permettre de comprendre comment les concepts et techniques actuels prolongent ou remplacent ceux qui étaient présentés il y a quelques années.

Dans la construction des programmes, nous nous intéressons principalement aux informations bien qu'on ne puisse les dissocier totalement des traitements effectués sur elles et

nous abordons successivement les modes de description des données, la notion de type de données et les environnements de construction de programmes.

2.1 DE LA STRUCTURATION DES TRAITEMENTS A CELLE DES DONNEES

2.1.1 Au commencement étaient les chaînes de bits...

Au début des années 50, programmer, c'était entrer en machine des suites de 0 et de 1 qui étaient interprétées comme des calculs à faire sur d'autres suites de 0 et de 1 ; on aura reconnu là l'absence de séparation entre données et programmes au niveau de la codification interne des informations dans un ordinateur. En 1988, le CRAY1 traite des millions de chaines de bits à la seconde, mais les mathématiciens, physiciens ou biologistes qui lui soumettent leurs calculs ne se soucient guère de ce niveau de représentation de leurs programmes et de leurs données. Les langages de programmation se sont imposés comme moyen de communication entre l'homme et la machine.

Entre temps, la programmation s'est "structurée" [WIR,71], [DAH,72], [ARS,77], on s'est appliqué à la pratiquer avec "méthode" [DIJ,76] et [JACK,75], [FIN,79], [PAI,79], [SCHO,79], elle est devenue science [GRI,81] ; parmi les références que nous venons de citer figurent les titres - symboliques - d'ouvrages qui, parmi tant d'autres, ont fait date dans le développement des idées en programmation et nous ont inspiré cette trilogie [ARS,87]: Structured Programming [DAH,72], A discipline of programming [DIJ,76], The Science of Programming [GRI, 81].

Nous montrons dans les trois paragraphes suivants comment cette marche continue vers davantage de structuration et d'abstraction dans les modes d'expression utilisés pour décrire des programmes a concerné d'une part les traitements, d'autre part les données, et comment elle s'est concrétisée de façons différentes mais convergentes dans des écoles de recherche et des domaines d'application différents.

2.1.2 La structuration des programmes

Un programme, comme le dit Wirth dans le titre de son ouvrage Algorithmes + Data Structures = Programs [WIR,76], comporte la description de traitements effectués sur des données. Très vite, il a fallu dépasser le tout numérique et les informaticiens se sont dotés de modes d'expression des traitements plus lisibles et plus structurés, les instructions des langages assembleurs, l'expression de calculs en FORTRAN, les structures de contrôle aujourd'hui classiques d'Algol 60. Pour les données, on a manipulé globalement des représentations d'entiers, de réels, de caractères, on les a regroupées en tableaux (FORTRAN, COBOL), fichiers (FORTRAN,COBOL), articles (COBOL,PL/1)...

Avec le développement d'applications nouvelles et l'accroissement de la puissance des machines, on a construit des programmes de plus en plus gros portant sur des volumes de données de plus en plus importants et il devenait hasardeux de travailler sans méthode et sans pouvoir vérifier la validité du programme produit. C'est ainsi qu'est né tout un courant de recherches sur les preuves liées aux structures de contrôles marqué notamment par les travaux de Floyd [FLO,67] et de Hoare [HOA,69] décrits dans [LIV,78] et sur la programmation structurée [DAH,72].

Pourtant malgré ces travaux théoriques et ces méthodes au niveau des structures de contrôle, on n'apportait pas de réponse satisfaisante aux concepteurs de gros programmes dont le principal problème était de décrire des traitements relativement simples sur de grandes quantités de données.

2.1.3 L'organisation des données

Ce sont les informaticiens de gestion qui ont été les premiers confrontés à ces difficultés, c'est donc dans ce domaine que sont nées les premières méthodes pour structurer des programmes à partir des données ou des résultats [WAR,72], [JAC,75], et qu'on a décomposé le travail de construction d'un programme en analyse fonctionnelle qui ne se préoccupait pas des fichiers physiques contenant les informations manipulées, puis en analyse organique et enfin en codage dans un langage de programmation; les deux premières étapes avaient essentiellement pour but d'organiser les informations à traiter dans des fichiers et de décrire les traitements à effectuer dessus.

Toujours dans l'informatique de gestion qu'on a appelée ensuite d'organisation, la recherche de solutions plus générales à la représentation et à l'interrogation de données de plus en plus complexes et interdépendantes a abouti aux travaux sur les bases de données avec les modèles réseau, hiérarchique et relationnel (Codd) et les langages de manipulation associés [DAT,75].

Dans les langages impératifs de nouveaux modes de structuration des données apparaissaient : les types énumérés, les ensembles en PASCAL et une nouvelle notion, cette de classe en SIMULA 67 [BIR,77]. Plusieurs auteurs proposaient de formaliser la notion de structure de donnée ou d'informations [EAR,71], [PAI,71]. Parallèlement, des travaux de l'école plus théorique de la programmation structurée et des preuves de programmes, émergeait en 1974 la notion de type abstrait de donnée implantée pour la première fois dans CLU [LIS,77].

Enfin, une troisième "école" de l'informatique développait à partir de 1960 des applications dites "d'intelligence artificielle" qui ont eu la caractéristique de vouloir d'abord faire faire aux machines des raisonnements difficiles, par exemple de la démonstration de théorèmes [NEW,63]. Mais très vite, la question des raisonnements est apparue seconde par rapport à l'immense quantité de données qu'il fallait fournir à une machine pour qu'elle ait, dans un domaine donné, un comportement "intelligent".

De ce courant de recherches sont issus les systèmes experts et leurs bases de connaissances [FAR,85], [LAURI,86]. Nous aurons l'occasion au cours de cet ouvrage de voir concrètement que le problème de la structuration de ces connaissances sur le plan des concepts et des outils n'est pas encore résolu de façon satisfaisante aujourd'hui.

Dans tous les domaines, l'intérêt s'est donc déplacé de l'aspect contrôle et traitement vers l'aspect modélisation d'univers et d'objets [BROD,84], [ABB,87] et les outils offerts ont quitté le niveau des machines pour aller vers celui des problèmes.

2.1.4 L'abstraction pour maîtriser la complexité et améliorer la fiabilité.

Il nous semble intéressant de résumer dans le schéma ci-après ces quelques trente années d'informatique qui, à travers plusieurs chemins, nous ont conduits aux modèles abstraits de données.

Dans tous ces domaines, à travers tous ces courants de recherches, l'objectif est de pouvoir définir des applications de plus en plus complexes et de produire des logiciels de plus en plus sûrs. Travailler sur des données abstraites est l'une des clés pour y parvenir. Mais produire du logiciel veut dire aussi utiliser les langages de programmation existants, c'est pourquoi nous consacrerons le paragraphe suivant aux notions de type et de modèle de données dans les langages.

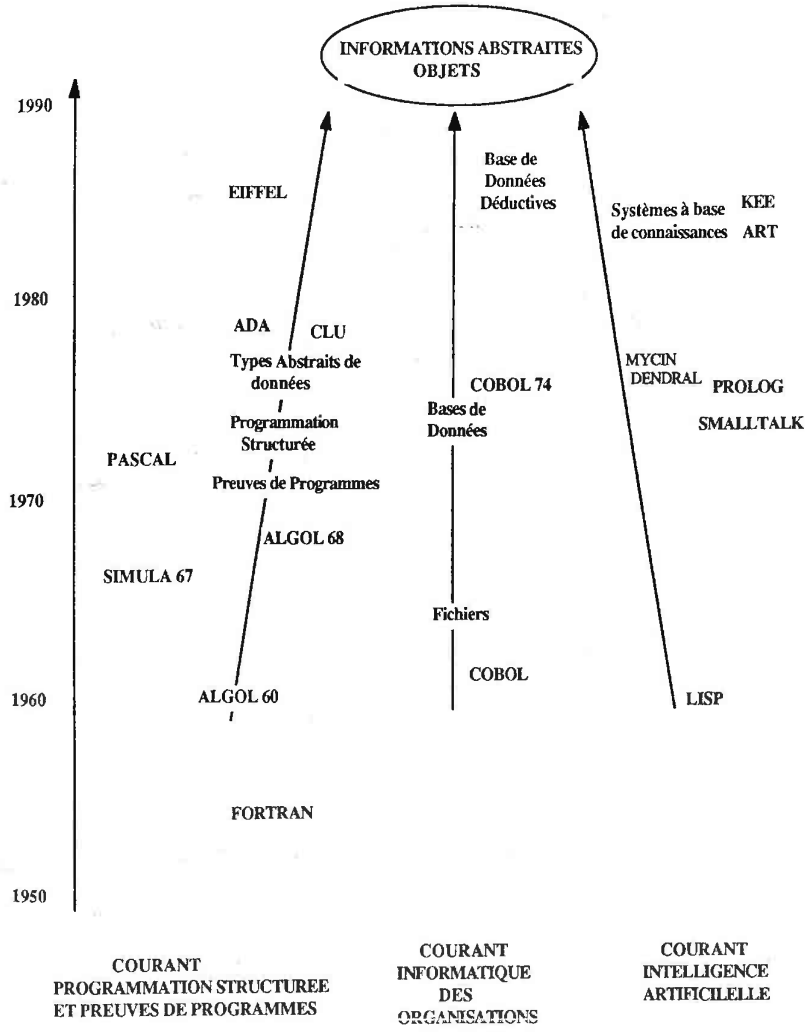


Figure 1 : Sur les chemins de l'abstraction

2.2 LES TYPES DE DONNEES

2.2.1 La notion de type dans les langages impératifs usuels

Intuitivement, la notion de type est associée à celle d'un ensemble de valeurs [DONA,85], pourtant si l'on y regarde de plus près, le type d'une variable dans un langage comme FORTRAN fournit trois sortes de renseignements de nature fort différentes :

- Il donne l'ensemble de valeurs que peut prendre tout objet du type.
- Il indique les opérations que l'on peut faire sur ces objets.

Ces deux informations sont utiles à l'utilisateur et un compilateur vérifiera que l'utilisateur a bien respecté les règles fixées ou du moins certaines d'entre elles.

- Il associe implicitement aux objets du type un mode de représentation en mémoire et donne ainsi au compilateur un renseignement sur la façon de réaliser une implantation des objets du type.

La plupart des langages offrent des mécanismes de typage qui rencontrent ces trois aspects et prennent les formes suivantes :

- des types de base, en général booléen, caractère, entiers, réels,
- des constructions de type : énuméré, intervalles,
- des modes de structuration d'objets de même type (tableaux, ensembles, fichiers) ou de types différents.

Ces possibilités pour typer des objets , associées à une demande de typage fort (pour permettre beaucoup de vérifications statiques) ne permettent pas de décrire des algorithmes à un niveau d'abstraction suffisant ; il y manque notamment la possibilité pour l'utilisateur de définir ses propres constructeurs de type et de les paramétrer.

2.2.2 Les objets des problèmes et des langages de haut niveau

Nous venons d'examiner la question des types du point de vue des langages, nous voulons maintenant la présenter du point de vue des problèmes [DAH,72] ou des langages orientés problèmes, ou des langages de haut niveau [JOR,77]. Que remarquons nous ?

- En mathématiques, beaucoup d'objets sont typés dans des énoncés de la forme: "soit f une fonction, g un groupe, t un triangle, etc..." et on associe ainsi à chaque objet formel un ensemble de valeurs, des opérateurs entre objets du type ou de types différents et des propriétés.

- En SETL [DEW,82], les objets sont des ensembles, des applications, des tuples.
- En Z [ABR,78], langage non implanté mais développé pour définir des spécifications, on utilise également le modèle ensembliste.
- Avec SIMULA 67 [BIR,77] était apparu le concept de classe permettant de regrouper un ensemble et les opérations s'y rapportant ; ce concept se retrouve dans la famille des langages dits orientés objets avec la notion d'héritage et dans les types abstraits de données.
- CLU [LIS,86], ALPHARD [WUL,75], ADA [BARNES,84] offrent la possibilité de donner des définitions abstraites d'objets et d'en définir séparément l'implantation .

Par rapport à ce que recouvre la notion de type dans les langages usuels, il apparaît d'abord qu'on retrouve au niveau des problèmes les idées d'ensembles d'objets et d'opérations sur les objets du type alors que l'aspect représentation en machine n'apparaît plus.

2.2.3 Les types abstraits de données

"Les types abstraits de données permettent de définir et structurer les ensembles d'objets et les opérations associées apparaissant dans l'énoncé d'un problème" [DER,79]. Ils formalisent la notion intuitive d'ensemble de valeurs et d'opérations associées évoquée aux paragraphes précédents [LIS,74],[GUT,77]. On constate un foisonnement de littérature consacrée aux types abstraits de données, et des différences, voire des divergences, qui, au delà d'une unanimité sur les concepts d'ensembles d'objets et d'ensembles d'opérations, proviennent surtout de cadres formels différents et de niveaux de formalisation inégaux [BID,81].

Dans le prolongement des deux voies explorées pour la vérification des programmes et décrites notamment dans [LIV,78], la voie axiomatique (Hoare, Floyd) et la voie point fixe (Scott, Strachey). Deux approches sont principalement développées pour des types abstraits de données : une approche axiomatique [WUL,75] mise en oeuvre notamment dans ALPHARD et une approche algébrique [GUT,77], [GOG,78].

De ces travaux théoriques sont issus des systèmes d'interprétations de spécification (AFFIRM) et des langages comme ALPHARD, CLU, ML utilisés pour la construction de prototypes, mais qui n'ont pas connu de diffusion hors des laboratoires de recherche, ainsi que des produits commercialisés comme ADA et EIFFEL. ADA est le seul parmi eux à proposer aujourd'hui ces mécanismes tout en bénéficiant d'une norme internationale ; EIFFEL [MEY,87b] élargit son cercle d'utilisateurs et compte parmi eux des industriels.

2.2.4 La modélisation de données dans d'autres contextes

Ce panorama ne serait pas complet si nous n'évoquions d'autres formes de programmation dans lesquelles la notion de type au sens où nous venons de la présenter n'intervient pas. Que fait-on en effet lorsqu'on travaille en PROLOG, en LISP ou dans une base de données relationnelle ?

. En PROLOG, on utilise des variables, des constantes, des prédicats qui permettent de construire des axiomes ; il s'agit donc d'un modèle unique sur lequel on n'exprime pas d'opérations.

. En LISP, tout est atome ou liste et on applique toutes les fonctions disponibles dans l'environnement à toute liste désignée comme argument ; les "données" et le "programme" jouent à cet égard le même rôle.

. Dans une base de données, on a un modèle unique (entités, relations) pour décrire l'univers du travail.

Les langages objets avec leurs classes tendent à offrir des mécanismes proches du typage, mais ne permettent que peu de contrôles ; certains essayent de rajouter des types à l'un ou l'autre des modèles précédents et proposent par exemple un PROLOG typé [MYC,84], [HSI,84], [BRAN,86].

La notion de type alliée à celle d'abstraction apparaît bien fondamentale dans la plupart des activités de modélisation d'univers en vue de la construction d'un programme. Elle intéresse différentes phases du cycle de vie d'un logiciel, mais elle n'est que l'un des outils à intégrer dans des environnements plus vastes que nous décrivons maintenant.

2.3 LES ENVIRONNEMENTS DE DEVELOPPEMENT DE LOGICIELS

2.3.1 Modèles de données et phases du cycle de vie d'un logiciel

La tendance aujourd'hui est au développement d'ateliers intégrés de génie logiciel comme Emeraude [BOUR,86] permettant d'apporter confort de travail et assistance au programmeur dans toutes les phases de construction et de maintenance de son produit.

Plusieurs projets dans ce cadre utilisent particulièrement les descriptions abstraites de données et peuvent être classés en deux catégories :

- ceux qui permettent une spécification assistée ou une transformation assistée de spécifications [BID,87],

- ceux qui s'appuient davantage sur la notion de module pour programmer en réutilisant des composants.

On retrouve l'avantage d'une définition modulaire et abstraite des données dans les procédures de maintenance et de modification des gros programmes, mais nous nous limitons ici à donner quelques exemples de réalisations dans les deux catégories précitées.

2.3.2 La construction assistée

Asspro (**Ass**istance à la **pro**grammation) [GRES,84], [SCHL,84], [BID,85], [BID, 87] est un environnement expérimental interactif et intégré pour faire de la construction de programmes.

"L'utilisateur, à partir d'un cahier des charges non formel, doit expliciter les structures de données intervenant dans cette description en donnant leur spécification algébrique écrite en PLUSS, si elle ne figure pas déjà dans la bibliothèque de spécifications. Pour cela, il dispose de ASSPEGIQUE (**Ass**istance à la **Spé**cification **algébrique**), un sous-ensemble d'outils pour créer et manipuler des spécifications de types abstraits algébriques et assurer la gestion de la bibliothèque de spécifications ...
L'étape suivante consiste à implanter dans le langage de programmation cible les structures de données identifiées. Un outil spécifique, Spada (traduction de spécifications en Ada) assiste et automatise partiellement cette implantation sous forme de paquetages Ada ...
La construction du programme proprement dit est ensuite effectuée sous le contrôle de Caty (Construction **a**utomatique de programmes à partir de **ty**pes abstraits) qui à partir de la description informelle du programme à construire, des spécifications algébriques de structures de données et des schémas de décomposition qui leur sont associés assiste et automatise le processus de construction à l'aide d'un certain nombre de stratégies ... "

L'utilisateur dispose ensuite d'un évaluateur symbolique et d'un générateur automatique de tests. Cet exemple nous semble significatif des tendances actuelles en ce domaine.

Parmi d'autres systèmes à base de spécifications développées ou en cours de développement, on peut citer OBJ [GOG,83], Affirm [GER,81], le projet CIP [CIP,81], le système de Feather [FEA,82], Larch, LPG, Oasis, le projet SACSO [LEV,86],[LEV,87] ,et parmi les environnements de programmation SPRAC, BVLISP [WER,84] ,Adèle [ADE,82]. Plus largement, ces applications sont elles-même construites dans des environnements conviviaux comme UNIX, INTERLISP.

2.3.3 La réutilisation de composants

Les environnements que nous venons de citer ont tous pour but d'augmenter à la fois la productivité du programmeur et la qualité du logiciel produit. Une façon d'atteindre ces objectifs est à l'évidence de réutiliser des "composants" : on ne refait pas le travail de construction et on bénéficie de la "sûreté" du composant existant [GOG, 86].

La réutilisation fait actuellement l'objet de proposition à deux niveaux :

Au niveau des spécifications

Nous venons de voir dans le projet ASSPRO une réutilisation limitée aux informations contenues dans la base du système (spécifications algébriques, schémas de décomposition et d'implantation en ADA). Le projet SACSO d'aides à la construction et la validation de spécifications prévoit des opérateurs de structuration pour favoriser la réutilisation de spécifications [LEV,87].

Au niveau des programmes

Le projet MAIDAY propose une aide au programmeur qui veut réutiliser un texte déja écrit en interprétant cette action comme la construction d'un fragment de programme guidée par la construction du fragment réutilisé.

De façon beaucoup plus pragmatique et sans méthode associée, G. BOOCH propose dans "Software Components whith Ada " [BOO,87] une bibliothèque de programmes ADA réutilisables portant sur les structures de données et des algorithmes usuels. Le cahier des charges du langage ADA comprenait d'ailleurs un environnement de développement de logiciels.

2.3.4 Vers la formalisation et l'expression d'une expertise en programmation ?

Assister le programmeur, prendre en charge une part de son travail suppose du coté de la machine une part d'expertise en développement de logiciel. Dans les environnements que nous venons de citer, on trouve des bases des connaissances ; de nouvelles sociétés qui commercialisent des produits de développement de logiciel comme "Reasoning Systems" avec REFINE opposent une "approche traditionnelle" et une approche "base de connaissances". Il nous semble en réalité qu'il faut aujourd'hui distinguer deux niveaux d'assistance dans les environnements proposés et à venir.

D'une part, il s'agit de mettre à la disposition de l'utilisateur de plus en plus de facilités de travail à la fois sur le plan du nombre de facilités offertes (larges bibliothèques de composants, de schémas, accès facile, outils de recherche, d'archivage, etc...), de la variété

des facilités offertes (dictionnaires, programmes, schémas à remplir, abréviations, génération de fragments de code, détection de redondances, d'incohérences..., spécifications exécutables, générateurs de tests...) et de l'intégration de ces nombreuses fonctionnalités (multifenêtrage, ergonomie bien pensée...). Dans ce cadre, l'expertise reste au programmeur qui prend la décision d'utiliser tel ou tel outil et de les enchaîner entre eux.

Il s'agit d'autre part de faire effectuer par la machine une part des tâches réservées jusque là au programmeur [HIL,87], parce qu'elles nécessitent des raisonnements qui ne sont pas réductibles à l'application d'algorithmes connus, notamment syntaxiques. Dans cette catégorie, nous trouvons le "Programmeur Apprentice" [RIC,78], un vaste projet qui comporte notamment une banque de plans de développement de programmes et dont PECOS [BARS,79a], [BARS,79b], LIBRA [KAN,83], KBEmacs (Knowledge Based Emacs) sont des étapes de développement. Un autre rapport général réalisé en 1983 au Kestrel Institute [GRE,86] reprend l'idée d'une assistance à partir de bases de connaissances tout au long du cycle de vie d'un logiciel mais n'a pas donné lieu à réalisation. Sur un point particulier du cycle de vie, le passage de spécifications algébriques à du code LISP, Raulefs [BART,81], [BART,82] propose un système à règles de productions formalisant les connaissances en programmation nécessaires à la réalisation de cette tâche.

Il apparait donc clairement que de nouveaux progrès en développement de logiciel viendront de systèmes sachant non seulement offrir de nombreuses possibilités à leurs utilisateurs [BARS,84a&b] mais sachant guider intelligemment [FIS,85], [EMB,87] ceux-ci au travers de leurs multiples ressources et prendre totalement en charge une part grandissante de leur travail.

Conclusion

Nous retiendrons de ce chapitre que le développement de logiciel commence nécessairement aujourd'hui par une description du problème à résoudre à un niveau d'abstraction élevé et qu'il faut donc créer et utiliser des modes d'expression permettant ce niveau d'abstraction et entraîner des futurs utilisateurs à travailler dans ces cadres.

Il est également évident que la construction de programmes quitte le stade artisanal pour entrer pleinement dans l'ère industrielle automatisée ; ce qui suppose l'utilisation d'environnements de travail perfectionnés qu'il faudra surtout savoir piloter.

Une part du processus de construction de programmes a déjà été automatisée à partir de travaux essentiellement syntaxiques, il faut étudier maintenant quelle nouvelle part de ce processus peut l'être à l'aide de techniques alliant syntaxe et autres connaissances, c'est l'objet du chapitre suivant.

Chapitre 3

VERS L'AUTOMATISATION DE LA PROGRAMMATION

Dès la fin des années 60, Waldinger et Lee publiaient "PROW : a step toward automatic programming" [WAL,69], ouvrant ainsi un large courant de travaux sur l'automatisation de tout ou partie du processus de construction de programmes parmi la communauté des chercheurs en intelligence artificielle.

Dans leur préface à un recueil d'articles sur le thème "Intelligence Artificielle et Ingénierie du Logiciel" [RICH,86], Rich et Waters notent que l'on peut s'intéresser à l'interaction entre ces deux domaines selon au moins deux points de vue : D'une part, les techniques issues de l'intelligence artificielle sont prometteuses pour aider le concepteur et le réalisateur de logiciel dans ses tâches en améliorant notamment sa productivité, son confort et la qualité du produit fabriqué, d'autre part, la construction de programmes s'est révélée être un domaine très fécond pour la recherche en intelligence artificielle, qui a motivé des développements nouveaux en représentation des connaissances et des raisonnements.

Plusieurs synthèses ont déjà été publiées sur le sujet [BAL,73], [BARS,84], [BIE,84], [BAL,85]. Notre objectif dans ce chapitre n'est pas d'en faire une nouvelle, mais simplement de prolonger la réflexion du chapitre précédent : quelles connaissances utilise-t-on pour formaliser le processus de programmation ? Comment représente-t-on un univers de travail ? A quelles phases du processus de construction peut-on s'intéresser dans une perspective d'automatisation ?

Nous montrons dans un premier paragraphe que la façon dont a été abordé le problème de la programmation automatique a beaucoup varié au cours des années, nous décrivons ensuite quelques projets qui nous paraîssent représentatifs des travaux en cours dans ce domaine et proposons, à partir de ces repères, une classification des connaissances utilisées.

3.1 UN VIEUX REVE : AUTOMATISER LA PROGRAMMATION

Qu'entend-on par "automatiser la programmation" ? En 1958, un tableau intitulé "automatic programming systems" est publié dans les communications de l'ACM, son contenu fait sourire aujourd'hui car il cite surtout des assembleurs et quelques compilateurs ! Nous nous plaçons délibérément à un niveau beaucoup plus proche des problèmes que les compilateurs des années 60 et définissons l'automatisation de la programmation comme la possibilité pour un utilisateur de définir son problème sous une forme commode et usuelle pour lui et d'obtenir automatiquement un programme qui résolve ce problème.

Nous n'entrons pas dans le débat " faut-il un programme ou seulement la solution du problème ?" [LAURI,78] et faisons l'hypothèse que nous voulons produire des programmes. Comme ce problème dans sa généralité est loin d'être résolu, il a fallu le simplifier et n'en aborder que certains aspects.

3.1.1 Simplifier le problème

La simplification peut porter sur :

- La liberté laissée dans le mode de description du problème .

Beaucoup de systèmes travaillent à partir d'un énoncé très formel [KAN,77], [BAR,79b], [WAL,79], alors que les recherches sur l'utilisation d'une langue quasi naturelle [RUT,76], [HEI,76] se sont quasiment détachées de ce courant.

- Le domaine du travail.

Certains systèmes génèrent des programmes dans des domaines limités ; les systèmes Draco [NEI,84] et Φnix [BARS,85b] sont les applications les plus élaborées dans cette voie puisqu'elles ont pour objet de générer des programmes dans des domaines particuliers mais tendent à être paramétrées par des caractéristiques du domaine.

- Le degré d'automatisation.

L'idée ici est de n'apporter qu'une automatisation partielle de la tâche à accomplir et de laisser à la charge de l'utilisateur par exemple des chaînes de transformations à appliquer. C'est la voie choisie par le "Programmer's Apprentice" [RIC,78]. Un des avantages de cette approche est qu'elle permet de construire des systèmes de plus en plus perfectionnés à partir d'un noyau initial, en augmentant au fur et à mesure le nombre de tâches prises en charge.

3.1.2 Choisir des techniques de travail

Une fois le problème simplifié, il fallait choisir des techniques permettant de passer d'une spécification à des lignes de code-programme.

Une première idée a consisté à appliquer à la construction de programmes des techniques de démonstration de théorème. On parle alors de synthèse de programme à partir d'une preuve constructive à chaque étape de laquelle est associée un pas du programme à créer [MAN,71], [MAN,75], [MAN,79], [MAN,80], [BAR,82]. Plus récemment, les travaux de Hayashi [HAY,86], Krivine [KRI,87] ou de Galmiche [GAL,88] vont dans le même sens. Mais la démonstration du théorème est elle-même une technique difficile, aussi les programmes produits par ce moyen sont-ils de petite taille et on s'oriente dans ce domaine aussi vers des systèmes de démonstration interactifs.

Une autre idée consiste à appliquer à une spécification du problème des suites de transformations formelles permettant d'aboutir à un programme et assurant tout au long du processus la conservation des propriétés du premier énoncé. Cette approche est fondée sur les travaux de transformation de programme illustrés notamment par Burstall et Darlington [BUR,77], [DAR,81] ; elle est utilisée dans [PAR,86] et dans le projet SPES [ALE,88] ; c'est aussi l'idée qui est à la base du projet PSI [GREE,77], [GREE,78a], [BARS,79a,79b] [KAN,83], [BARS,85b]. On y trouve des systèmes complètement automatiques parce qu'ils n'ont à chaque étape que peu de transformations applicables et des systèmes où la sélection des transformations reste manuelle.

Selon les auteurs, l'accent est mis sur la conservation de la correction à travers l'application des différentes transformations et sur la possibilité d'avoir des preuves formelles [BRO,81] ou simplement sur la formalisation d'une expertise en programmation sous forme de règles [BARS,79b].

3.1.3 Les résultats obtenus

Dans les synthèses complètement automatiques de programmes, les exemples traités sont souvent des "exercices d'école" comme dans le texte qui suit et ils sont résolus au prix de l'insertion dans le système de nombreuses règles :

Exemple donné par Manna et Waldinger dans [RIC,86] :

Enoncé: Calcul de la racine carrée entière d'un entier non négatif .

Spécification :

sqrt (n) <== trouver z tel que entier (z) et z^2 <= n < (z + 1)^2

où entier (n) et 0 <= n

Schéma général applicable :

f (a) <== trouver z tel que R (a, z) où P (a).

La technique de démonstration utilisée consiste à supposer que P (a) est vraie et à prouver (et donc ici construire) qu'il existe z tel que R (a, z).

Cependant certains systèmes visent des problèmes de taille plus significative, nous en donnons quelques exemples au paragraphe suivant .

3.2 QUELQUES REALISATIONS SIGNIFICATIVES DANS CE DOMAINE

Nous présentons successivement des travaux issus du projet PSI (Stanford) qui représentent des systèmes entièrement automatiques, les recherches du Programmer's Apprentice (MIT) où est prévue une intervention d'utilisateur et enfin une réalisation européenne d'implantation de types abstraits algébriques [BART,82].

3.2.1 Le système PECOS (D.Barstow, 1979)

L'objectif de ce travail était d'expérimenter et de voir dans un domaine particulier s'il était possible de formaliser les connaissances du programmeur pour les faire utiliser par une machine. Le domaine de travail choisi est l'implantation d'algorithmes abstraits portant sur des nombres, des listes, des relations et utilisant des structures de contrôle classiques sur ces objets (parcours séquentiel "pour chaque", conditionnelles, itérations).

PECOS construit son programme par raffinements successifs à partir de l'algorithme initial . Parmi les connaissances qu'il utilise, les trois-quarts concernent des techniques générales de programmation, notamment des règles de représentation d'objets abstraits, et le dernier quart est spécifique au langage LISP. Voici un exemple d'algorithme pour lequel PECOS peut choisir des représentations de données et produire un programme LISP :

Le sujet traité est un petit problème de classification dans lequel on donne un ensemble nommé CONCEPT dans le texte et une série d'autres ensembles nommés SCENE. Pour chaque SCENE, on veut savoir s'il compte parmi ses éléments tous ceux du CONCEPT. PECOS reçoit la description suivante:

Data structures

```
CONCEPT  a collection of integers
SCENE    a collection of integers or QUIT
```

Algorithm

```
CONCEPT <-  input a list of integers ;
loop :
      SCENE <- input a list of integers or the string QUIT ;
      if SCENE = "QUIT" then exit the loop ;
      if CONCEPT is a subset of SCENE
            then output the message "fit"
            else output the message "didn't fit" ;
      repeat;
```

Les différents programmes produits diffèrent par la représentation choisie pour SCENE. Le test d'inclusion CONCEPT ⊂ SCENE est transformé en une énumération des éléments de CONCEPT et pour chacun d'eux un test d'appartenance à SCENE par une règle d'implantation de ce test.

Dans le cas le plus simple SCENE et CONCEPT sont tous deux représentés par des listes chainées, c'est la représentation standard dans le système à l'issue de la saisie de données. Mais d'autres règles suggèrent que :

- on peut modifier la représentation d'un objet pour optimiser un traitement effectué dessus.
- le test d'appartenance est plus rapide sur une liste triée que sur une liste chainée simple
- le test d'appartenance est plus rapide avec une représentation de l'ensemble par un tableau de booléens .

Et PECOS a successivement proposé pour ce problème des listes triées, des tableaux de booléens, des "property lists" de LISP et des sous-listes accédées avec une fonction de hachage, en appliquant ces règles générales relatives à des choix de représentation d'objets.

Comme exemple de règle spécifique à la production de code LISP, on peut citer : "Si une liste chainée est représentée par une liste du langage LISP sans emplacement réservé pour la représentation de la tête, l'accès au premier élément de la liste peut être implanté par un appel de la fonction CAR".

Les règles sont en fait représentées comme dans l'exemple suivant :

Règle : Un ensemble d'élément courant x peut être représenté par une liste chaînée et une référence à ce même élément.

Représentation de la règle :

```
(REFINE (SEQUENTIAL - COLLECTION
        ( GET - PROP ELEMENT (BIND X)))
        ( NEW - NODE LINKED - LIST
        (SET - PROP ELEMENT X )))
```

La forme générale d'une telle règle est :

(REFINE <node pattern> < refinement specification >)

Toutes ces règles portent sur des transformations de faible envergure et on obtient le programme final à la suite de l'application de nombreuses règles, ce qui ne permet pas de suivre facilement le raisonnement d'ensemble opéré par PECOS. Mais ce n'était pas l'objectif du travail, de même que l'efficacité n'était pas non plus visée a priori ; PECOS applique toutes les règles qu'il peut appliquer et génère plusieurs programmes pour un même algorithme.

Le problème du choix entre plusieurs implantations est évidemment crucial pour l'efficacité du programme produit. Dans le projet PSI, il y a en fait deux composants qui interviennent au niveau de la génération de programme : l'un, PECOS, construit un arbre d'implantations possibles pour l'algorithme, l'autre, LIBRA, explore cet espace de recherche et fait des choix en évaluant les efficacités relatives des différentes implantations.

3 . 2 . 2 Le système LIBRA (E. Kant) [KAN, 83]

Ce système, complémentaire du précédent et appartenant au même projet, s'attache à choisir des implantations d'algorithmes efficaces parmi celles qui sont développées dans PECOS au travers de l'arbre de raffinement d'algorithme dans lequel autant de noeuds fils sont créés qu'il y a de règles applicables à un noeud père donné.

LIBRA utilise à la fois des techniques d'analyse algébrique d'algorithmes et une base de connaissances sur des critères de choix de représentations. Entre deux solutions radicales, les représentations standards pour chaque objet abstrait qui aboutissent à des programmes inefficaces et la recherche de la représentation optimale qui serait très coûteuse et demanderait des techniques beaucoup plus fines et complexes que celles utilisées ici, LIBRA propose des solutions efficaces.

Voici un exemple de programme et d'arbre des coûts développé par LIBRA ; on propose une banque d'histoires représentées part des mots-clés, puis à la donnée d'un mot-clé quelconque, on veut que le programme affiche la liste des histoires le contenant.

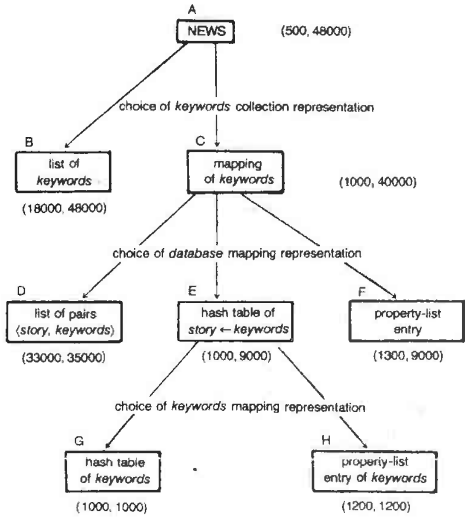
Data Structures

database : mapping from *story* to *keywords* ;
story : string ;
keywords : collection of *key* ;
key : string ;
command : alternative 'xyzzy' or *key* ;

Algorithm

```
input (database) ;
loop
 repeating
    input (command) ;
    if command = 'xyzzy' then assert-exit quitloop ;
    forall S  in domain-of(database) when command  in database [S] do output(S);
 exits
    quitloop ;
 end;
```

Arbre des coûts



Environ 400 règles permettent d'envisager des représentations et de combiner des fonctions d'évaluation qui supposent que l'utilisateur a fourni des informations telles que le nombre d'histoires dans la base de données (80), le nombre moyen de mots-clés par histoire (100), le nombre d'itérations (300) et la probabilité pour qu'une commande soit un mot clé d'une histoire (0.01).

Plus encore que PECOS, LIBRA, qui utilise des fonctions de coût déterminées de façon heuristique et statistique, n'est pas prévu pour expliquer les choix effectués et le raisonnement conduit. Il représente cependant une des meilleures tentatives d'automatisation de choix efficaces d'implantation de données abstraites.

3.2.3 Le "Programmer's Apprentice " [RIC,78]

L'objectif de ce projet est de créer un environnement de développement de logiciel dans lequel le système assiste le programmeur aussi souvent que possible. Le P.A. repose sur les concepts de plan pour représenter un programme et d'une bibliothèque de plans pour construire et modifier des programmes ; il comprend cinq grandes parties : un analyseur qui établit un plan à partir de la description de l'algorithme, un grapheur qui donne une visualisation du plan retenu, une bibliothèque de plans pour des morceaux d'algorithmes classiques et un éditeur de plans qui doit inciter le programmeur à travailler sur les plans plutôt que sur les textes de programme, notamment pour les modifications.

Si l'idée de plans et de composition de plans est séduisante, il faut remarquer que les plans utilisés sont très fragmentaires, se situent à de bas niveaux syntaxiques et les auteurs reconnaissent qu'il faudrait une bibliothèque de plusieurs milliers de plans pour écrire la plupart des programmes simples qui sont le lot quotidien des programmeurs. Une autre limite de cette conception est qu'elle concerne essentiellement les structures de contrôle et ne prend pas en compte la représentation des données.

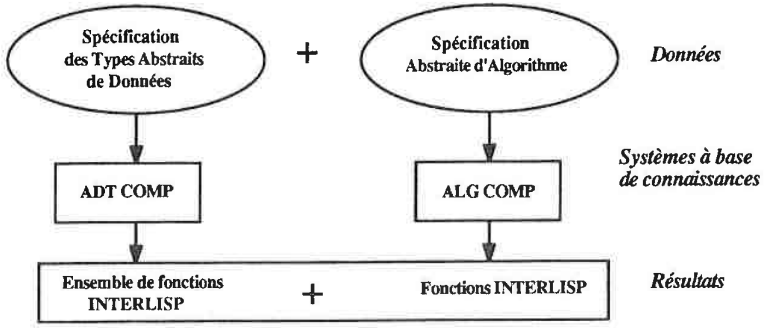
Ces limites étant posées, KBEmacs, l'un des logiciels mettant en oeuvre les principes du Programmer's Apprentice permet de construire, sous Emacs et en utilisant une base de connaissances reposant sur le principe des plans, des programmes, notamment en Ada à partir de "cadres" prédéfinis ou en LISP.

3.2.4 APE [BART,82]

APE est un système expert en programmation LISP recevant pour données des spécifications abstraites de types de données et les algorithmes portant sur ces données. Il a un objectif beaucoup plus limité que le système précédent, il serait plutôt de la même catégorie que

celui de Barstow (PECOS) puisqu'il s'appuie sur une formalisation de règles de programmation dans une base de règles de production. Son originalité réside dans l'utilisation de spécifications algébriques de données et donc dans l'expression des connaissances permettant de passer de telles spécifications à des implantations de ces types en LISP et dans la séparation entre la base de connaissances permettant d'implanter les types de données et celle permettant de coder les algorithmes. Cela a évidemment pour conséquence que les caractéristiques de l'algorithme ne sont pas prises en compte pour le choix de représentation des données.

De nombreux programmes LISP ont pu être produits par ce système qu'on peut résumer dans le schéma suivant :



3.3 QUELLES CONNAISSANCES POUR AUTOMATISER LA PROGRAMMATION ?

Tous les systèmes que nous venons de décrire ou de citer ont en commun de posséder des connaissances en programmation ; notre objectif dans ce paragraphe, est de faire une rapide analyse du type de connaissances possédées et de la façon dont elles sont représentées.

3.3.1 Typologie des connaissances

Une première catégorie de connaissances est constituée par des règles de programmation que nous qualifierons de "bas niveau" qui permettent d'obtenir du code LISP à partir de certaines situations intermédiaires ; ces règles concernent l'implantation de structures de contrôle classiques et la représentation de données [BARS,79&82].

A un niveau au dessus, nous trouvons des connaissances nécessaires pour piloter l'application des précédentes [KAN,83]. Enfin nous trouvons plus rarement des stratégies plus générales de développement de programme comme "diviser pour régner", "recherche binaire" et des schémas de programmes.

Globalement, ces systèmes renferment beaucoup d'informations parcellaires, concernant une petite partie du programme ou une donnée et peu d'informations globales sur l'algorithme ou le programme. On a aussi beaucoup plus d'informations statiques et syntaxiques que d'informations dynamiques sur les données ou le comportement à l'exécution.

Ces connaissances concernent la phase de création de programme proprement dite ; les systèmes plus ambitieux d'assistance à la production de logiciel ont à l'évidence besoin d'autres connaissances : où aller chercher une information à quel moment ? Quand suggérer de réutiliser un composant, quand créer ? Cette catégorie de connaissances n'est pas encore formalisée.

3.3.2 Représentation des connaissances

Selon les systèmes, ces connaissances sont exprimées sous forme de règles de transformation de programmes, de règles de raffinement, de schémas de programmes ou de plans.

Les règles ont donné des résultats relativement significatifs mais elles sont rarement accompagnées d'outils de vérification.

Les schémas prennent eux-mêmes des formes variées : schémas formels, unités génériques de langages, cadres à remplir [WAT,85], [BID,87],dessins.

La notion de plan combinant schémas et spécifications formelles est plus riche et moins limitative, elle permet une vision plus globale des stratégies mais ne figure que dans le Programmer's Apprentice. Elle semble également refléter mieux que d'autres l'image mentale des connaissances du programmeur expert [SOL,84].

Nous concluons de ce rapide tour d'horizon que, si tous les auteurs s'accordent sur les voies dans lesquelles il faut s'engager, les réalisations sont encore de faible envergure par rapport aux projets annoncés. D'autres recherches ont nécessité de formaliser des connaissances en programmation ; en effet, les systèmes d'enseignement de la programmation doivent contenir une part d'expertise dans ce domaine ; les solutions retenues peuvent compléter le panorama ci-dessus. Quelques réalisations sont décrites dans le chapitre suivant consacré de façon plus générale aux logiciels pédagogiques.

Chapitre 4

ENSEIGNEMENT DE L'INFORMATIQUE ET INFORMATIQUE DANS L'ENSEIGNEMENT

L'ordinateur peut être un outil pédagogique et nous nous proposons de l'utiliser pour enseigner un cours d'informatique. Il nous semble donc pertinent de rappeler dans ce chapitre les principales applications de l'informatique dans l'enseignement [BES,82], [MIL,82], [O'SHEA,83], [LEF,84], [HER,85], [DiSES,87], [MAD,87], [FOR,87] ; nous accordons une attention particulière aux voies qui essaient de répondre aux besoins non encore satisfaits en s'appuyant sur les progrès les plus récents de la technique informatique.

Nous analysons ensuite quelques caractères propres à l'enseignement de l'informatique et nous nous attachons enfin à décrire les réalisations qui appartiennent à la même catégorie que notre projet : les systèmes d'aide à l'enseignement de l'informatique.

4.1 UTILISATION DE L'INFORMATIQUE DANS L'ENSEIGNEMENT

La pénétration de l'informatique dans la plupart des secteurs d'activité, la baisse du coût des matériels, la disponibilité de nombreux logiciels sur micro-ordinateurs amènent à modifier les contenus et les méthodes d'enseignement dans beaucoup de disciplines. Nous examinons ici les principales formes d'utilisation de l'informatique dans l'enseignement afin de créer des repères pour apprécier sur ce point la situation de la discipline informatique.

Nous faisons le choix, malgré leur intérêt certain, de ne pas développer les aspects transdisciplinaires, ni les apports sur le plan de la gestion administrative ou pédagogique.

4 . 1 . 1 L'ordinateur pour résoudre des problèmes dans une discipline.

Nous employons ici le terme résolution de problèmes dans un sens très large puisque nous classons dans cette rubrique aussi bien les traceurs de courbes et autres imagiciels du cours de mathématiques que le pilotage d'expériences au laboratoire de sciences physiques, les simulations, la consultation de banques de données ou l'utilisation de traitements de textes. L'ordinateur est dans ce cas utilisé dans la classe de la même façon qu'il le serait au bureau, à l'usine ou au centre de recherches ; il permet d'obtenir des informations, des résultats pour répondre à des questions posées et résoudre des problèmes.

Les élèves se voient ainsi proposer des sujets d'étude qu'il n'était pas possible d'aborder avant par manque de moyens de calcul ; ils doivent en contrepartie se familiariser avec de nouvelles démarches de résolution de problème, attacher plus d'importance à l'organisation d'un travail, à sa décomposition en tâches élémentaires, qu'à la mise en oeuvre de techniques calculatoires ou de recherche documentaire par exemple que la machine peut prendre en charge.

Les logiciels utilisés sont parfois adaptés au contexte pédagogique d'une classe, mais sont de même nature que ceux utilisés par les professionnels ; ces outils nouveaux peuvent entraîner dans la classe des modes de travail renouvelés .

4 . 1 . 2 La séquence d'enseignement guidée par ordinateur

Nous préférons ce titre à celui d'Enseignement Assisté par Ordinateur pour bien marquer que nous nous intéressons là aux séquences de formation, courtes ou longues, s'adressant à des individus isolés ou à des élèves avec leurs maîtres, guidées par ordinateur à l'aide de logiciels dédiés à l'enseignement.

Les premiers programmes, purement linéaires, ont pris le relais de l'enseignement programmé développé sur papier ; peu à peu, les progrès technologiques (écrans couleur, crayons optiques, souris, son, etc...) et des conceptions plus élaborées ont donné des systèmes plus conviviaux et mieux adaptés à leur utilisateur. On les construit aujourd'hui avec méthode et des milliers de logiciels de ce type sont commercialisés.

Ces systèmes ont pour avantage d'appeler une participation permanente de l'élève et donc de maintenir son attention sur le sujet de son étude, de pouvoir solliciter ses facultés d'apprentissage de façon variée (lecture de textes, observation d'images ou de graphiques, résolution de problèmes, exercices de mémorisation...), de respecter son rythme de travail et donc de permettre aux élèves d'une classe de travailler à des vitesses différentes sur des sujets

différents ou à des individus isolés d'apprendre pendant leurs moments de liberté.

Mais ils sont encore relativement pauvres, en ce sens qu'ils ne peuvent comprendre aucune question spontanée de l'élève, ne lui font résoudre que des exercices dont la solution est pré-enregistrée et ne peuvent que faiblement s'adapter aux réactions d'un individu donné. Ils visent plus souvent à transmettre ou à vérifier des connaissances qu'à développer ou contrôler un raisonnement.

4 . 1 . 3 Les environnements pour l'enseignement

Nous entendons par environnements pour l'enseignement à la fois ceux où l'élève est libre d'explorer le champ des possibilités offertes par l'environnement qu'on lui propose, l'univers LOGO [PAP,80], les jeux d'aventures appartiennent à cette catégorie, et ceux où il est "intelligemment" guidé par le système [HAR,73], [SLE,82], [QUE,85], répertoriés sous le terme de tuteurs intelligents [NIC,88].

La première catégorie vise à offrir des activités supposées stimulantes pour le développement de facultés intellectuelles générales telles que l'imagination, la capacité de réaction devant des situations variées, etc...

Les tuteurs intelligents [DED,86], [YAZ,86a&86b], [BARON,87], [SLE,87], on parle aussi parfois d'EIAO, veulent combler les lacunes de la première génération de didacticiels. Ils ont souvent une base de connaissances et un système de déduction qui leur permet de résoudre eux-mêmes les problèmes qu'ils posent aux élèves, d'expliquer le raisonnement qu'il font pour parvenir à la solution [SAF,85], [KAS,86], de détecter les erreurs les plus classiques commises par les apprenants et d'adapter la conduite du dialogue au comportement de l'utilisateur grâce à un modèle de l'activité d'apprentissage de ce dernier.

Cette approche très prometteuse peut aujourd'hui être développée en bénéficiant à la fois des performances des matériels mais aussi des recherches effectuées en intelligence artificielle sur la modélisation des raisonnements et des connaissances, et notamment sur la modélisation du raisonnement humain. Elle suppose un large effort de recherche en didactique des disciplines, afin que soit peu à peu formalisée une expertise pédagogique [COR,84], [VIV,87] pas assez rigoureusement exprimée aujourd'hui pour être utilisée dans des systèmes automatisés, et en psychologie cognitive afin de mieux comprendre les processus d'apprentissage des utilisateurs [AND,84], [MAT,85] et de pouvoir les représenter dans un modèle de l'élève [PAL,88].

Peut-on utiliser l'ordinateur pour enseigner l'informatique ? Avec laquelle des

approches indiquées ci-dessus ? Avant de répondre à ces questions, il nous faut nous interroger sur ce qu'est l'enseignement de l'informatique.

4 . 2 L'ENSEIGNEMENT DE L'INFORMATIQUE

4 . 2 . 1 L'ordinateur objet et outil d'enseignement

L'informatique a déjà ceci de particulier par rapport à d'autres disciplines : l'ordinateur et les logiciels que nous nous proposons d'utiliser comme véhicules de savoir et de savoir faire sont ici d'abord objets d'enseignement. Cette situation n'est pas unique, on la retrouve avec l'enseignement de la langue maternelle par exemple, mais elle mérite attention.

Cela peut-être une chance car les départements d'informatique des écoles et universités sont bien équipés en matériels performants, les élèves sont à l'aise avec le maniement des machines, les enseignants savent créer ou adapter des logiciels et peuvent donc réaliser plus facilement que d'autres des outils pédagogiques.

Mais cela suppose que cette dualité soit bien prise en compte ; toute utilisation de machine et de logiciel pour introduire ou illustrer des concepts d'informatique a des "effets de bords" sur le cours qui est par ailleurs consacré aux machines ou aux logiciels ainsi utilisés, autant tenir compte de façon positive de ces interactions dans la phase de conception et d'articulation des différents enseignements.

4 . 2 . 2 Des contenus en évolution rapide

Une autre caractéristique de l'informatique est que c'est une science jeune, encore en train de se constituer en tant que science et très dépendante de techniques matérielles et logicielles. Ces changements techniques permanents font que d'une part une large fraction des enseignements leur est inévitablement consacrée et d'autre part les enseignants, déjà trop peu nombreux, passent beaucoup de temps à dépouiller des notices nouvelles pour en faire des présentations pédagogiques ; lorsque ce travail de pionnier est achevé, une réflexion didactique plus poussée pourrait conduire à l'élaboration de didacticiels, malheureusement de nouvelles notices arrivent... dont les contenus rendent caduques une part des réflexions précédentes.

Un tel contexte n'est pas favorable à l'élaboration de traditions pédagogiques stables et d'outils les mettant en oeuvre. Il faut cependant nuancer ce tableau un peu sombre : d'une part certains domaines fondamentaux de l'informatique, notamment au niveau des enseignements de base, ont acquis une stabilité suffisante pour que le développement

d'instruments pédagogiques assurés de la durée y soit possible, d'autre part les nouveaux outils du génie logiciel devraient permettre de développer plus rapidement des didacticiels aux fonctionnalités plus agréables, plus variées et mieux adaptées aux différents besoins.

4 . 2 . 3 Qu'enseigne-t-on ?

Les enseignements d'informatique incluent des cours sur les circuits logiques, les composants et périphériques d'un ordinateur, les commandes des systèmes, la syntaxe des langages... toutes choses qu'il faut apprendre !

L'informaticien doit non seulement avoir acquis des connaissances mais il doit surtout savoir les utiliser dans la résolution de problèmes variés. Il doit acquérir des méthodes pour vaincre la complexité des situations qui lui sont proposées, savoir non seulement travailler, mais aussi communiquer, dialoguer avec l'utilisateur du système qu'il construit.

Et on sait très bien que, s'il est facile de contrôler les connaissances d'un programmeur en syntaxe des langages - les compilateurs font cela très bien - on n'apprend vraiment à programmer qu'en résolvant des problèmes variés. Il existe aujourd'hui des méthodes de construction de logiciels, JACKSON [JAC,75], SADT [SCHOM,82], MERISE [TAR,83] ; elles ont été conçues dans l'ère du "papier-crayon", elles sont difficiles à enseigner, là encore il faut "pratiquer" ; elles deviendront peut-être plus faciles à utiliser sur les stations de travail intelligentes que nous évoquions au chapitre précédent... un effort demeure nécessaire pour la phase de l'apprentissage de ces méthodes.

4 . 2 . 4 Formation générale et formation professionnelle

Enfin, l'informatique est enseignée à des publics différents avec des objectifs différents. Il s'agit dans certains cas de former des professionnels, dans d'autres de donner une culture générale au citoyen d'une société informatisée.

Pour la formation de professionnels, il faut employer chaque fois que c'est possible des outils réellement utilisés dans la profession ; la question est alors "d'entourer" ces outils d'une "coquille" pédagogique qui en permette une prise en main progressive et assistée. Ces coquilles pédagogiques s'adressent à la fois à un public de formation initiale et aux professionnels en formation continue.

Pour l'enseignement général, il s'agit d'initier les élèves à un certain nombre de concepts et de techniques, mais pas de les rendre opérationnels sur un outil donné. Des environnements proprement pédagogiques peuvent être construits à cet effet.

4.3 L'ORDINATEUR OUTIL PEDAGOGIQUE DANS L'ENSEIGNEMENT DE L'INFORMATIQUE

Nous avons vu que l'enseignement de l'informatique revêtait en fait de multiples facettes. En ce qui concerne la programmation, une synthèse présentant les principaux logiciels dédiés à l'enseignement de la programmation est donnée dans [duBOU,86] et les perspectives d'avenir sont analysées dans [duBOU,87b]. Mais l'enseignement de l'informatique ne se réduit pas à celui de la programmation [PAS,87] même si ce dernier a été le sujet de la majorité des réalisations.

Nous donnons maintenant quelques exemples d'utilisation de l'ordinateur pour enseigner l'informatique, représentatifs des différentes approches existant actuellement.

4.3.1 Enseignement assisté par ordinateur de type tutoriel

Les logiciels de ce type actuellement commercialisés pour l'enseignement de l'informatique portent sur :

- le fonctionnement interne d'un ordinateur (unité centrale), ses composants, ses périphériques, MICADO, ses logiciels, INITIAL, FASSY,
- les langages de commande, notamment UNIX avec FORM-UNIX et MSDOS avec TUTORIAL SET,
- la syntaxe de langages de programmation, par exemple BASIC avecBASICANIME ou C avec FORM-C,
- la prise en main de progiciels par exemple FRAMEWORK II EAO, ou APPAT pour Multiplan.

Ces logiciels se présentent sous la forme d'une alternance de séquences tutorielles (dérivées de l'enseignement programmé) et de simulations plus ou moins graphiques.

4.3.2 Apprentissage de la programmation pour les débutants

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 réalisations motivantes. C'était l'une des idées de base de LOGO, plus récemment des logiciels comme AGD ou MINOS offrent des cadres pour l'initiation.

Si l'on envisage ensuite l'apprentissage de la programmation dans un langage

largement répandu, on trouve d'une part des environnements offrant des fonctionnalités et des moyens de visualisation particuliers, APILOG de Softia et POPLOG [GIB,84],[du B,85] en sont des exemples pour l'apprentissage de la programmation logique, d'autre part des environnements qui dépassent le cadre purement syntaxique et essaient soit de guider les élèves dans la construction d'un programme, soit plus simplement de vérifier la validité du programme construit par rapport au problème qui était posé.

Beaucoup de tentatives on été faites dans ce domaine depuis MYCROFT [GOL,75] qui détecte et corrige des erreurs dans des programmes LOGO ; elles concernent des débutants [KOF,75], [LAUB,81], [MIL,82], [AND,86] et sont donc limitées à la prise en compte de petits programmes.

La conception de tels systèmes a été motivée par une forte demande, il fallait initier beaucoup de gens à la construction de programmes et on ne disposait que d'un petit nombre d'enseignants. Elle s'explique aussi par la nature essentiellement "créative" et "constructive" de l'activité de programmation qui ne peut s'apprendre que par la pratique ; les tâtonnements et erreurs sont nombreux chez les débutants et il était donc naturel qu'on espère une aide de la machine dans ces phases de démarrage. Enfin, l'activité de "résolution de problème" que constitue la construction d'un programme a rapidement intéressé les psychologues [HOC,78] et leurs études devraient aider à proposer dans ce domaine des modèles de l'élève [SOL,81], [SOL,82], [SOL,83a&b] .

Il existe une famille des systèmes de "vérification de programmes", ils ont un modèle du problème ou de sa solution et travaillent à partir de ce modèle.

LAURA [ADA,80] reçoit un squelette du programme à produire et utilise des propriétés sur les graphes pour décider si le texte FORTRAN fourni par l'étudiant est équivalent au modèle.

Deux autres systèmes PROUST [JOH,85 a&b] et LISP tutor [AND,85] sont commercialisés depuis plusieurs années. PROUST (Program Understanding for Students) est capable de détecter et d'expliquer 70% des erreurs rencontrées dans les programmes PASCAL produits par les débutants. Il est cependant nécessaire de lui fournir une description "adaptée" des problèmes qu'il soumet aux étudiants ; malgré cette restriction, la version originale de PROUST se compose de 15 000 lignes de LISP sur VAX 750 ; la version commercialisée tourne sur PC mais reconnait un ensemble d'erreurs beaucoup moins riche. Le LISP tutor travaille avec des règles de définition et de codification de fonctions LISP, environ 300, et des règles de "mauvaise codification", environ 450.

Un autre système, BIP, assiste l'étudiant dans la construction de programmes BASIC, sans analyse profonde des programmes eux-mêmes, mais avec aussi une banque d'exercices pré-déterminés.

Ces dernières années, est apparue une nouvelle génération de systèmes ; ils sont dotés de connaissances leur permettant d'analyser n'importe quel programme fourni par l'étudiant (il y a certes des limites mais nous opposons cette catégorie à la précédente où on ne travaille qu'avec des banques d'exercices pré-enregistrés).

RADAR (Reasoning on ADA Rubbish) recherche et corrige des erreurs de nature sémantique dans des programmes ADA ; il est destiné à des programmeurs ayant l'expérience d'un langage comme COBOL, FORTRAN ou PASCAL et qui débutent en ADA. Il est basé sur une "méta-description" des programmes . Le "Pascal Program Checker" de M. Elsom Code [COO,86] essaie de présenter les erreurs détectées selon la perception qu'un débutant peut en avoir et non selon le point de vue du compilateur.

CAPRA [GAR,86], [FER,88] est un autre projet de tuteur en programmation. Enfin deux systèmes récents et ouverts concernant l 'apprentissage de LISP méritent d'être décrits.

PHENARETE [WER,85] détecte et corrige des erreurs de débutants en LISP en "comprenant" les programmes ; elle a été écrite à la suite d'une étude systématique des erreurs observées chez les élèves et d'une classification de ces erreurs ; elle suit le travail de l'utilisateur tout au long de la construction des programmes, propose des corrections ou des améliorations dès qu'elle en détecte la possibilité. Ce système est effectivement utilisé avec des étudiants de l'université Paris VIII autour de BVLISP [WER,84].

LISP-CRITIC [FIS,87] dispose d'une banque de règles lui permettant de proposer des améliorations à la version LISP fournie par l'utilisateur, en expliquant y compris par des schémas sur les listes pourquoi la représentation proposée est équivalente à la précédente.

4.3.3 Aides à la prise en main d'un outil

Cette catégorie de logiciels, déjà citée en 4.3.1, nous semble devoir tenir une place grandissante et mérite donc un paragraphe spécifique. En effet, ils sont partie intégrante d'une politique de commercialisation de nouveaux produits, ils présentent des caractères spécifiques que nous illustrons sur des exemples et, si l'on doit en construire en grand nombre, il faut s'intéresser à en systématiser le développement.

Nous avons insisté sur l'importance d'une assistance de ce type dans la formation

initiale et continue des professionnels de l'informatique. Ces environnements ne sont arrivés que récemment sur les marchés mais tendent à se développer aujourd'hui notamment pour favoriser l'apprentissage autonome de nouveaux logiciels. Le lancement d'un nouveau produit accompagné de son logiciel d'apprentissage peut en effet être un plus pour un diffuseur de logiciel, l'obstacle d'une nouvelle formation à assurer au personnel étant souvent un frein au changement de produits dans les entreprises.

Un exemple intéressant de cette famille est APPAT [BRO,87]. Dans APPAT, commercialisé aujourd'hui sous le nom d'HYPERCALC, on introduit l'utilisateur dans l'environnement de MULTIPLAN, en lui faisant résoudre au fur et à mesure, des exercices pré-enregistrés lui permettant de découvrir les différentes fonctions disponibles. La composante pédagogique de ce système a été produite après analyse des concepts à faire acquérir aux débutants dans ce type de logiciels ; elle analyse de façon simple le comportement de l'apprenant et lui propose une progression d'exercices adaptés aux difficultés détectées.

Les travaux concernant la conception et la réalisation de tels systèmes progressent actuellement dans deux directions. D'une part, on cherche à contrôler les actions de l'utilisateur et à lui fournir des explications en cas de manoeuvre erronée, comme dans les environnements PHENARETE et LISP-CRITIC, d'autre part, on voit apparaitre des générateurs de systèmes d'autoformation à un progiciel comme LINE ou STARGUIDE [CLAE,88]...

Il faudrait cependant se demander dans ce domaine ce qui doit être spécifique à un environnement d'enseignement, et ce qui devrait à terme figurer dans tous les environnements intelligemment assistés d'utilisation de logiciels [BRE,87].

4.3.4 Autres applications

La plupart des travaux que nous venons de citer s'adressent à des débutants en informatique. Il existe cependant quelques tentatives dans des domaines différents comme :

- un expert assistant l'utilisateur dans la pratique de la méthode Jackson [SCW,87],

- la conduite de projets en génie logiciel [COU,86 a&b]. IPHIGENIE tente de modéliser l'activité du concepteur de logiciel, d'apprendre aux étudiants à décomposer un projet en étapes, à faire à chaque étape des choix obéissant aux contraintes de leur environnement, le raisonnement est guidé par un système expert.

4.3.5 La question des raisonnements et des méthodes

Globalement, les environnements existants contrôlent peu le raisonnement fait par le programmeur aux différentes étapes de son travail et ne le guident pas dans le processus de création d'une application ; si des progrès sont en cours sur le plan du contrôle comme nous venons de le mentionner, il reste beaucoup à faire sur le plan des méthodes pour construire des applications.

Dans ce dernier domaine, MAIDAY [GUY,84] est un environnement de construction d'algorithmes basé sur une méthode de construction déductive de programmes [PAI,79], [DUC,84].

Il nous semble que c'est dans ce secteur des méthodes et des raisonnements que les manques sont actuellement les plus grands. Les experts informaticiens savent-ils suffisamment expliciter leur façon de travailler pour que ce savoir soit mis à la disposition d'élèves dans des systèmes d'enseignement ? Ce n'est pas sûr. Dans le domaine des représentations d'informations abstraites, notre objectif est bien de réaliser un environnement permettant d'entraîner l'élève aux raisonnements à effectuer et de lui faire acquérir une démarche de travail; la réalisation de cet objectif nécessite d'abord une étape de formalisation de ces raisonnements. Des études de même nature doivent être entreprises dans d'autres domaines.

LES TYPES ABSTRAITS DE DONNEES ET LEURS REPRESENTATIONS : ETUDE D'OBJECTIFS, DE CONTENUS ET DE FORMES D'ENSEIGNEMENT

II° Partie

Introduction à la partie II

Nous consacrons cette seconde partie à l'étude de la matière à enseigner à travers un choix d'ouvrages représentatifs de l'évolution de ces enseignements dans le temps et de la diversité des publics et des approches pédagogiques. Cette étude vient compléter et étayer notre propre expérience d'enseignement dans ce domaine.

Un premier chapitre décrit les objectifs, les contenus et les présentations proposées dans ces ouvrages. Un second chapitre est consacré plus particulièrement au sujet pour lequel nous proposons un logiciel d'assistance, l'implantation de types abstraits usuels dans les langages algorithmiques. Enfin, on ne définit pas des objets indépendamment des algorithmes dans lesquels ils sont utilisés, nous abordons donc dans le chapitre 3 la question du mode d'expression des algorithmes et des programmes. Nous montrons comment on peut utiliser les concepts avancés implantés en ADA pour écrire des procédures sur des objets abstraits et les compléter pour aboutir à des programmes exécutables efficaces.

Les ouvrages utilisés contiennent la part d'expertise sur les critères d'implantation d'objets qui est formalisée, nous en montrons les limites. Ils doivent nous servir de point de repère pour les contenus à proposer dans le système que nous définissons ; nous concluons donc le chapitre 1 sur des tableaux récapitulatifs de contenus.

Chapitre 1

OBJECTIFS ET CONTENUS DES MANUELS

Gerbier dans "Mes premières constructions de programmes" [GER,77] insistait sur la nécessité d'une progression pédagogique associant structures de données et structures de contrôle tout au long de l'enseignement de la programmation. Il semble y avoir consensus dans la plupart des formations en informatique pour un cours de base sur la construction des algorithmes et des programmes ; dans ce cours, les applications traitées, tout en associant étroitement données et traitements, ne mettent pas en oeuvre des données nécessitant des études fines de structuration et de représentation.

Vient ensuite un module nommé selon les cas "Structures de Données", "Programmation Abstraite" ou "Algorithmique Avancée" dans lequel sont exposées de façon plus systématique, les notions relatives aux types de données, à leurs représentations, à l'analyse de l'efficacité des programmes produits. Ce sont des ouvrages pouvant illustrer un tel module que nous analysons dans ce chapitre.

1.1 OBJECTIFS GENERAUX

1.1.1 Principes

Si nous excluons certaines publications qui se contentent de présenter les structures de données disponibles dans un ou plusieurs langages (tableaux, enregistrements, fichiers) et n'offrent pas d'intérêt dans le cadre de cette étude, la plupart des auteurs ont, autour des notions de spécifications et d'implantations de types de données, des objectifs voisins qui sont bien illustrés par les extraits de préface ou d'introduction ci-après :

Horowitz et Sahni [HOR,76]

"En résumé, nous avons essayé de mettre l'accent pour nos étudiants, sur

- l'aptitude à définir à un niveau d'abstraction suffisamment élevé les structures de données et les algorithmes nécessaires,
- l'aptitude à imaginer des implantations différentes pour une structure de données,
- l'aptitude à écrire un algorithme correct,
- l'aptitude à analyser le temps de calcul nécessaire à l'exécution du programme résultat. En plus, deux autres objectifs, non explicitement prioritaires au départ, sont atteints aussi : le premier concerne la notion d'algorithme "bien structuré"... Nous espérons qu'en lisant des programmes écrits dans un bon style, les étudiants prendront de bonnes habitudes. Le second concerne le choix des exemples. Nous avons essayé de choisir des exemples qui illustrent bien une notion, qui ont des applications en informatique...".

Gaudel, Soria, Froidevaux [GAU,86]

"Ce livre présente les types de données et les algorithmes fondamentaux, dont la connaissance est indispensable à tout informaticien, en développant de façon accessible les résultats récents dans ce domaine.

Le volume 1 est consacré aux fondements théoriques nécessaires : représentation des données et notion de type abstrait, introduction à l'analyse de la complexité des algorithmes... les différents points traités sont illustrés par de nombreux exemples d'application.

Le volume 2 expose les principales méthodes de recherche, de tri et de traitement sur les graphes. Pour chacun des problèmes, on présente différentes représentations des données et les algorithmes correspondants. Les algorithmes proposés sont analysés du point de vue de leur complexité en place mémoire et en temps d'exécution, ce qui permet une étude comparative de ces méthodes.

Chaque chapitre est accompagné de très nombreux exercices."

Nous notons donc une volonté commune de développer chez l'étudiant :

- une aptitude à définir des types abstraits de données,
- une aptitude à écrire des algorithmes sur de tels types,
- une aptitude à imaginer des implantations différentes pour chaque objet abstrait et à comparer ces implantations du point de vue de leur efficacité sur le programme produit,
- une aptitude à écrire des algorithmes puis des programmes bien structurés,

en apportant comme connaissances :

- une définition de la notion de type abstrait,
- des représentations possibles,
- des notations et outils pour évaluer l'efficacité d'un algorithme,
- des exemples d'applications développées selon ces principes,
- des algorithmes que l'on retrouve dans beaucoup de problèmes informatiques.

Au delà de cette volonté commune, les présentations sont très diverses ; certaines différences s'expliquent par des considérations d'ordre historique dont nous donnons un aperçu maintenant.

1.1.2 Evolution dans le temps

Horowitz et Sahni écrivent dans la préface de leur ouvrage "Fundamentals of Data Structures" [HOR,76] :

"Il est fascinant et instructif de retracer l'histoire du contenu de ces cours. Vers le milieu des années 60, le cours ne s'appelait pas "Structures de données" mais peut-être "traitement de listes". Les thèmes principaux en étaient SLIP, JPL-V et SNOBOL.

Puis en 1968 apparut le premier volume de "The Art of Computer Programming" de D. Knuth. Sa thèse était que le traitement des listes n'était pas quelque chose de magique qui ne pouvait se faire qu'à l'intérieur d'un système spécialement conçu à cet effet. Au contraire, il montrait que les mêmes techniques pouvaient être mises en oeuvre dans presque n'importe quel langage et il déplaçait l'attention vers la construction d'algorithmes efficaces...".

Peu d'informaticiens français se reconnaitront dans ce passé parce qu'il n'y avait pas (ou si peu!) de cours de ce type lorsque l'ouvrage de Knuth est paru. Plus récemment, les enseignements déjà bien identifiés de "Structures de données" ont continué à évoluer comme l'indiquaient GAUDEL, SORIA et FROIDEVAUX dans leur préface :

"Cet ouvrage résulte de plusieurs années d'expérience de cet enseignement. Au cours des années, son contenu a peu évolué sur le fond : les algorithmes et les types de données sont restés les mêmes. Cependant, il a été sensiblement modifié quant à la forme en fonction de l'évolution des méthodes de spécification et de programmation et de l'introduction de nouvelles techniques d'analyse des algorithmes. Dans sa version présente, d'une part, nous insistons sur la nécessité d'une expression claire et non ambiguë des algorithmes, de leurs spécifications et des types de données utilisés ; d'autre part une analyse rigoureuse de tout algorithme, basée sur des outils mathématiques bien compris, nous paraît fondamentale".

C'est en effet dans la forme des présentations, dans la rigueur des formalismes

proposés pour définir les types, qu'on note le plus d'évolution. Toute une famille d'ouvrages [WIR,76], [BOUS,84], [LUC,83], [HOA,78], [COUV,84], [HOR,76], [AHO,83], [AHO,74], [WIR,86], [BOO,87] présente la notion de type abstrait de donnée informellement en citant les opérations principales effectuées sur des objets de ce type et en exprimant la "sémantique" de ces opérations par le texte d'une procédure dans un langage qu'on peut qualifier de pseudo pascal, c'est-à-dire par référence à une représentation d'un objet de ce type dans le langage.

Seuls quelques ouvrages récents parmi ceux destinés à l'enseignement [MAR,86], [GAU,86], [PAI,86], [PAI,88] et des textes de recherche [YEH,78], [LEV,84] donnent des spécifications formelles d'un type et de ses opérations avant d'en proposer des implantations.

Dans la suite du chapitre, nous prenons en compte les contenus de tous les ouvrages, mais nous n'étudions les questions de formalisme et de terminologie que sous leurs aspects les plus récents.

1.2 PRESENTATIONS INFORMELLE ET FORMELLE D'UN TYPE

1.2.1 Approches de la notion

Globalement, l'importance des objets caractérisés par un couple (ensemble de valeurs, opérations) est rappelée dans le processus de construction d'algorithmes par raffinements successifs. Les auteurs indiquent ensuite qu'ils vont présenter les principaux types d'objets rencontrés dans la construction des algorithmes en informatique. On trouve par exemple dans [PAI,86] : "Ce chapitre étudie rapidement les types les plus fondamentaux rencontrés en informatique, ou plutôt les constructeurs fondamentaux qui permettent d'obtenir des types à partir d'autres types : nous avons déjà rencontré liste, ensemble, fonction, arbre binaire. A leur propos, nous donnerons des indications générales sur les représentations des types".

Pour un type donné, on trouve soit une définition, soit une présentation intuitive par référence à un environnement connu ; en voici quelques exemples :

Pour le type LISTE :

Dans [PAI,86] "... La sémantique d'un tel type est la notion de suite finie d'éléments de V. Du point de vue algorithmique, on utilise une liste chaque fois qu'on effectue un traitement

séquentiel".
Dans [HOR,76] "Un des types de données les plus simples et les plus fréquemment rencontrés est la liste linéaire, ou ordonnée... .
Si nous considérons une liste ordonnée de manière plus abstraite, nous disons qu'elle est soit vide, soit qu'elle peut être écrite comme $(a_1,a_2,a_3...a_n)$ où les a_i sont des éléments d'un ensemble S".

Pour le type ENSEMBLE :

Dans [AHO,83] "L'ensemble est la structure de base sous-jacente à toutes les mathématiques. Dans la conception des algorithmes, les ensembles sont utilisés comme base de beaucoup de types abstraits de données... nous présentons le "dictionnaire" et la "file avec priorité", deux types abstraits de données basés sur le modèle ensemble...".
Dans [COU,84] "Un ensemble est une collection d'objets sans répétition, et sur le plan fonctionnel, il n'existe pas de relations entre les éléments d'un ensemble, si ce n'est qu'ils sont tous distincts ; la fonction de base est l'appartenance et lie chaque élément à la collection".

Pour le type GRAPHE :

Dans [GAU,86] "Beaucoup de problèmes de la vie courante, tels la gestion de réseaux de communication et l'ordonnancement de tâches correspondent à des structures relationnelles que l'on peut modéliser par des graphes. Informellement un graphe est un ensemble d'objets, appelés sommets, et de relations entre ces sommets."

Ce dernier extrait nous amène à la définition par des exemples qui est souvent utilisée. Que ce soit pour introduire un type ou pour illustrer une définition préalablement donnée, la question des exemples nous paraît importante.

1.2.2 Nature et rôle des exemples

Certains auteurs utilisent systématiquement beaucoup d'exemples empruntés à des situations variées pour présenter chaque type. Pour d'autres, les exemples ne viennent que pour des types considérés sans doute comme plus difficiles à justifier, c'est le choix de [GAU,86] qui ne donne pas d'exemples pour les listes, les piles et les files et qui en introduit pour les structures arborescentes.

Dans les ouvrages les exemples interviennent à deux niveaux :

On les trouve pour introduire ou illustrer une notion, c'est le niveau que nous venons d'évoquer; il est caractérisé souvent par une présentation informelle de l'exemple et par un problème présentant un objet bien identifié comme étant du type étudié.

Les exemples interviennent aussi dans la catégorie des exercices résolus ou proposés... et là, on a, nous semble-t-il, trop souvent des cas où il n'y a qu'un objet à représenter et où le choix d'un type pour cet objet ne fait pas de doute. C'est une bonne chose si l'on suit l'objectif de [HOR,76] "donner des exemples significatifs de l'utilisation d'un type", mais cela n'entraîne en rien à réagir devant bon nombre de situations concrètes où il y a beaucoup d'objets à définir et où les opérations effectuées sur ces objets ne conduisent pas de façon si évidente, à un choix de type.

1.2.3 Contenu et terminologie

Outre une approche intuitive et parfois une définition mathématique, la présentation d'un type de donnée comprend la liste des opérations disponibles sur les objets du type.

Ces opérations sont :

- soit décrites informellement et implantées par des procédures,

Exemple : [AHO,83]
"INSERER (x, p, l) insère l'élément x à la position p dans la liste l, déplaçant les éléments des places p et suivantes d'une position.
Ce qui veut dire que si l = a_1, a_2..., a_n, alors l devient a_1, a_2,..., a_{p-1}, x , a_p..., a_n.
Si p est FIN (l),alors l devient a_1..., a_n , x ;
Si la liste l n'a pas de position p, le résultat est indéfini.
PLACE (x, l) retourne la position de l'élément x dans la liste l ..."

- soit décrites formellement sans terminologie de référence
Exemple : [HOR,76]
Structure PILE (élément)

déclaration CREER ( ) ---------- pile
AJOUTE (élément, pile) ---------- pile
ENLEVE (pile) ---------- pile
SOMMET (pile) ---------- élément
ESTVIDE (pile) ---------- booléen

Pour tout s appartenant à pile, pour tout i appartenant à élément on a :

ESTVIDE (CREER)	= vrai
ESTVIDE (AJOUTE (i, s))	= faux
ENLEVE (CREER)	= erreur
SOMMET (CREER)	= erreur
SOMMET (AJOUTE (i, s))	= i

fin pour
fin structure PILE

- soit décrites formellement en référence à une terminologie dont l'usage se répand et qui met l'accent sur le rôle des différents composants de la spécification d'un type.
Exemple : [GAU,86]

Type abstrait :
sorte Pile
utilise Booléen, Elément
opérations

pilevide	: ----> Pile
empiler	: Pile x Elément ----> Pile
dépiler	: Pile ----> Elément
estvide	: Pile ----> Booléen

Les opérations sommet et dépiler ne sont définies que si la pile n'est pas vide et satisfont les axiomes suivants pour toute pile p et tout élément e :

sommet (empiler (p, e)) = e
dépiler (empiler (p, e)) = p

D'autre part, l'opération estvide vérifie :

estvide (pilevide) = vrai
estvide (empiler (p, e)) = faux

Cette spécification précise le type défini à l'aide du mot **sorte** ainsi que les types qui interviennent dans la définition de pile, à savoir ici Booléen et Elément ; suivent ensuite les profils des opérations définies pour ce type. Ensembles et opérations constituent la **signature** du type ou la forme syntaxique de sa définition.

Une liste de propriétés qui doivent vérifier les opérations sous forme d'axiomes en constitue la **partie sémantique**. On parle dans ce cas de **spécification algébrique** ou de **type abstrait algébrique.**

Toutes les opérations ne jouent pas le même rôle, on peut les classer suivant plusieurs critères :

Pair dans [PAI,86] distingue parmi les opérations :
- les générateurs qui permettent de créer les objets du type,
- les caractéristiques qui permettent d'utiliser les objets du type et sont liées aux générateurs par des axiomes.

Gaudel dans [GAU,86] distingue par ailleurs entre les opérations internes qui rendent pour résultat un objet du type défini et les observateurs qui rendent pour résultat un objet de l'un des types utilisés dans la signature.

De même, toute liste d'axiomes n'est pas acceptable pour définir complètement un type, on parle de "complétude suffisante" lorsque l'ensemble des axiomes permet effectivement d'engendrer les objets du type et de donner des valeurs au résultat de chaque opération définie.

1.3 LES TYPES PRESENTES

Nous récapitulons dans ce paragraphe sous forme de tableaux les types les plus couramment présentés ainsi que les opérations proposées dans la définition de certains types dans les ouvrages qui nous ont servi de référence pour cette étude.

1.3.1 Ouvrages utilisés

Traité ou manuel pédagogique ?

On trouve aujourd'hui beaucoup d'ouvrages publiés sur le sujet ; il faudrait y ajouter des documents moins largement diffusés mais qui ont permis l'élaboration de nombreux enseignements comme les documents de cours des écoles d'été de l'AFCET plusieurs fois consacrées à la construction de programmes, ainsi que les innombrables polycopiés universitaires. Nous avons étudié des documents dont la plupart étaient annoncés comme support de cours. Cependant, les uns sont présentés sous forme de traités avec un souci d'exhaustivité [HOR,76], [AHO,83] alors que d'autres veulent davantage illustrer certains concepts fondamentaux de la programmation [PAI,86].

Dans un traité, la matière est organisée par thème ; on y présente chaque type, avec pour chacun d'eux beaucoup de représentations, les unes très usuelles, les autres plus rares mais présentant un intérêt particulier par l'originalité de la représentation, de la performance de certaines opérations ou des études de complexité possibles. Le lecteur est supposé motivé, il vient y chercher des références, on n'a pas nécessairement besoin de lui proposer des

exercices facilitant l'assimilation progressive des notions présentées.

D'autres ouvrages sont des supports de cours moins complets, et en ce sens plus réalistes comme contenus d'enseignement ; ils ne décrivent que certains types et, pour chacun d'eux, quelques représentations permettant d'illustrer certains concepts. Les documents étudiés sont d'ailleurs rarement consacrés uniquement aux types de données et à leurs implantations ; ils abordent aussi d'autres aspects de la programmation. Dans [GAU,86], on trouve par exemple 80 pages sur 200 consacrées aux outils mathématiques nécessaires à l'étude de la complexité des algorithmes.

Des points d'entrée différents pour des objectifs et des publics différents.

Un cours a nécessairement des objectifs prioritaires mais aussi des objectifs seconds. En effet, d'une part on ne peut mutiplier les cours à l'infini, d'autre part, certaines notions sont naturellement reliées entre elles. Dans le domaine que nous étudions, la notion d'efficacité d'algorithme doit être présentée à propos d'applications où son intérêt se justifie, elle est donc naturellement liée à l'enseignement d'implantation de données permettant de proposer des algorithmes efficaces pour les opérations définies sur ces données.

Parmi les objectifs prioritaires rencontrés pour des cours, nous pouvons citer :

- l'entraînement à l'écriture d'algorithmes abstraits ; cet objectif est, nous l'avons vu, celui de nombreux ouvrages ; dans ce cas, l'étude de représentations nécessitant une programmation complexe au niveau des opérations définies sur le type est une motivation supplémentaire pour séparer les difficultés et ne se préoccuper de ces questions que lorsqu'une version complète de l'algorithme est écrite au niveau abstrait.

- l'analyse de la complexité des algorithmes et la présentation d'algorithmes originaux qu'on a peu de chances de réinventer et qu'il faut donc étudier.

Parmi les publics destinataires de telles formations, on trouve les étudiants des formations initiales en informatique et des professionnels en formation continue. Dans chacune des catégories, on peut viser une initiation donnant une idée de la façon dont on aborde des applications complexes, un entraînement à la programmation abstraite ou une présentation systématique des types de données et de leurs représentations en mémoire.

Le tableau du paragraphe suivant à propos des types présentés dans différents ouvrages doit être lu à la lumière de ces remarques.

1.3.2 Types et implantations proposées

Nous donnons ci-dessous, à titre d'exemples, trois tableaux :

- Le premier indique la référence complète des ouvrages utilisés pour ces comparaisons ; il est bien évident que nous avons cherché la diversité, mais que nous ne prétendons nullement à l'exhaustivité. Ces références figurent aussi dans la bibliographie générale, ainsi que celles d'autres ouvrages que nous n'avons pas nécessairement cités mais qui auraient pu également servir de support à ces tableaux.

- Le second indique pour chacun de ces ouvrages les types présentés. Il faut le compléter par une remarque sur l'ordre de présentation des types. Certains auteurs s'attachent à une structuration de l'ensemble des types. Gaudel les classe en structures séquentielles, structures arborescentes et structures relationnelles ; Finance, dans son cours de licence, préfère présenter les deux constructeurs de base ensemble et produit cartésien, puis, les suites, les listes avec comme cas particulier les piles et les files, les tables ou fonctions et enfin les arbres.

- Le troisième montre les opérations définies pour le type ensemble dans les environnements cités.

AHO A., HOPCROFT J., ULLMAN J.
Data Structures and Algorithms
Addison Wesley, 1983

BOOCH G.
Software Components with ADA : Structures, Tools and Subsystems
Benjamin/Cummings Pub. Comp., 1987

BOUSSARD J.C., MAHL R.
Programmation Avancée
Eyrolles, 1984

COUVERT A., PEDRONO R.
Techniques de Programmation - Cours C45 (Polycopié)
Université de Rennes, 1984

FINANCE J.P.
Notes de Cours de Licence Informatique
Université de Nancy1, 1985

GAUDEL M.C., SORIA M., FROIDEVAUX C.
Types de Données et Algorithmes, Vol. 1 et 2
Collection Didactique, INRIA, 1987

HOROWITZ E., SAHNI S.
Fundamentals of Data Structures
Pitman, 1976

LEVY N., SOUQUIERES J.
Spécification des Types et Opérations de Base dans le Système SACSO
Document Interne CRIN, 1986

LISKOV B., GUTTAG J.
Abstraction and Verification in Program Development (CLU)
MIT Press London, 1986

MARTIN J.J.
Data Types and Data Structures
Prentice Hall, 1986

PAIR C.
Programmation et Structures de Données (Polycopié)
Ecole des Mines de Nancy (INPL), 1986

SCHWARTZ J.T., DEWAR R.B.K., DUBINSKY E., SCHONBERG E.
Programming with Sets. An Introduction to SETL
Springer Verlag, 1986

Références des ouvrages utilisés

	Arbre	Ensemb.	File	Graphe	Liste	Pile	Séquen.	Table
Aho Hopcroft Ullman	oui	oui	oui	oui	oui	oui		
Booch								
Boussard Mahl	oui		oui	oui		oui	oui	oui
Couvert Pedrono	oui	oui	oui	oui	oui	oui		oui
Finance	oui		oui			oui		oui
Gaudel Soria Froidevaux	oui		oui	oui	oui	oui	oui	oui
Horowitz Sahni	oui		oui	oui	oui	oui		oui
Lévy Souquières		oui					oui	oui
Liskov		oui			oui		oui	
Martin	oui	oui	oui	oui	oui	oui	oui	oui
Pair	oui	oui	oui		oui	oui		oui

Types présentés

	Aho	Booch	Couvert	Lévy	Liskov	Martin	Pair	Schwartz
Vide	oui	oui	oui	oui	oui	oui	oui	oui
Adjonction	oui	oui	oui	oui	oui	oui	oui	oui
Indéfini				oui				oui
Ensemble des parties								oui
Ensemble des parties de cardinalité K								oui
Singleton			oui	oui				
Suppression	oui	oui	oui	oui	oui	oui	oui	oui
Union	oui	oui	oui	oui		oui		oui
Intersection	oui	oui	oui	oui		oui		oui
Différence	oui	oui	oui					
Union disjointe	oui							
Complémentaire			oui					
Copie	oui	oui	oui					oui
Différence symétrique			oui					oui
Elément de singleton				oui				
Cardinalité		oui	oui	oui	oui	oui		oui
Inclus ?		oui	oui	oui				oui
Appartient ?	oui	oui	oui	oui	oui	oui	oui	oui
Vide ?		oui	oui	oui	oui	oui		
Egal ?	oui	oui	oui	oui				oui
Non égal ?								oui
Disjoints ?			oui					
Minimum, maximum	oui							
Un quelconque			oui		oui			oui
Partie de P(s) dont x est un élément	oui							
Itération		oui	oui					oui

Opérations définies pour le type ensemble

Chapitre 2

REPRESENTATIONS D'UN TYPE

Nous avons présenté dans le chapitre précédent les objectifs et contenus de manuels, traités et articles traitant de la question des types abstraits de données.

Comme notre but n'est pas seulement d'écrire des algorithmes abstraits mais aussi de produire des programmes efficaces, nous nous intéressons dans ce chapitre au problème de la représentation d'un type abstrait à l'aide des types et opérations disponibles dans les langages impératifs usuels. Dans un premier paragraphe, nous définissons le problème de la représentation d'un type abstrait et abordons la question des niveaux de représentation. Nous étudions ensuite l'influence des représentations sur l'efficacité des programmes produits et présentons des critères de choix de représentation pour quelques types usuels.

Nous concluons sur quelques problèmes qui restent ouverts.

2.1 LE PROBLEME DE LA REPRESENTATION

De manière intuitive, représenter un type source par un type cible, c'est définir les objets et opérations du type source à l'aide d'objets et d'opérations du type cible en conservant les propriétés du type source chaque fois que c'est possible.

2.1.1 Les niveaux de représentation

Le type cible envisagé ci-dessus n'est pas nécessairement celui d'un langage de programmation. En effet, la conception d'un algorithme un peu complexe se fait toujours en plusieurs étapes correspondant à des raffinements successifs [WIR,76], [GAU,86] ; pour les informations traitées par l'algorithme, on a alors des niveaux d'abstraction successifs ; et il est important de pouvoir représenter de façon systématique les types définis à un certain

niveau par ceux définis à un autre niveau.

On peut vouloir par exemple représenter un type ENSEMBLE de CARACTERES par un type LISTE de CARACTERES ; il faut alors fournir pour chaque opérateur du type ENSEMBLE sa représentation en termes d'opérations du type LISTE. Lévy dans [LEV,84] étudie ce problème dans le cadre des types abstraits algébriques et propose des transformations pour passer de manière systématique d'un type à un autre.

Bien que ce problème soit important, notamment dans une perspective de réutilisation de bibliothèques de types existantes, nous nous limitons ici au cas où les types sources sont des types abstraits définis par un ensemble de sortes et d'opérations et les types cibles les types des langages de programmation.

2.1.2 La représentation d'un type

Reprenons l'ensemble du type PILE [GAU,86] décrit au chapitre 1 de cette seconde partie. La figure suivante redonne la description du type abstrait. L'auteur propose ensuite une représentation de ce type en Pascal, en passant en fait d'une spécification algébrique à une spécification constructive en termes de suite et en représentant cette suite par un tableau.

Type abstrait :

sorte Pile
utilise Booléen, Elément
opérations

pilevide	: ----> Pile
empiler	: Pile x Elément ----> Pile
dépiler	: Pile ----> Elément
estvide	: Pile ----> Booléen

Les opérations sommet et dépiler ne sont définies que si la pile n'est pas vide et satisfont les axiomes suivants pour toute pile p et tout élément e :

sommet (empiler (p, e)) = e dépiler (empiler (p, e)) = p

D'autre part, l'opération estvide vérifie :

estvide (pilevide) = vrai estvide (empiler (p,e)) = faux

Une représentation contiguë du type PILE et de ses opérations en PASCAL:

```
type PILE = record SOM : integer ;
                   ELTS: array [1... N] of char ;
end ;
```


La pile vide est représentée par tout **record** dont le champ SOM contient 0. Les opérations du type abstrait sont données dans la figure suivante :

```
procedure PILEVIDE (var P : PILE ) ;
begin
      P.SOM : = 0
end PILEVIDE ;

procedure EMPILER (var P : PILE ; X : char ) ;
begin if  P. SOM = N   then write  ('erreur pile pleine')
                       else begin P.SOM : = P. SOM + 1 ;
                                  P.ELTS [P. SOM] : = X
                            end
end EMPILER ;

procedure DEPILER (var P : PILE ) ;
begin if  ESTVIDE (P)  then write ('erreur pile vide' )
                       else P.SOM : = P. SOM- 1
end DEPILER ;

function SOMMET (P : PILE ) : char ;
begin if  ESTVIDE (P)  then write ('erreur pile vide')
                       else SOMMET : = P.ELTS [P.SOM ]
end SOMMET ;

function ESTVIDE (P : PILE) : boolean ;
begin if  P.SOM= 0     then ESTVIDE : = true
                       else  ESTVIDE : =  false
end ESTVIDE ;
```

Qu'a-t-on fait dans le programme ci-dessus ? On a représenté une pile par un article (déclaration "record") dont le premier champ contient le nombre d'éléments de la pile et le second un tableau de dimension suffisante pour contenir tous les éléments de la pile au sens de l'exécution de l'algorithme.

On a ensuite écrit des en-têtes de procédures traduisant les opérations définies sur le type pile en utilisant cette représentation. On a enfin choisi des algorithmes permettant d'implanter ces opérations et on les a exprimés en PASCAL.

Représenter un type abstrait, c'est donc :

- trouver une représentation des objets du type abstrait avec les types du langage cible,
- exprimer les opération définies sur le type en termes des types et constructions syntaxiques du langage cible,
- choisir pour implanter chaque opération un algorithme et le coder dans le langage choisi.

Il peut donc exister de nombreuses façons de représenter un type abstrait de données dans un langage de programmation ; des choix différents peuvent en effet être faits à chacune des étapes ainsi mises en évidences.

Voici à titre d'exemple une autre représentation du type PILE utilisant la technique du chaînage et des opérations associées :

objet :

```
type PILE = ^ ELT ;
ELT = record VAL: char ;
             LIEN: PILE
      end ;
```

opérations :

```
procedure PILEVIDE (var P : PILE ) ;
begin
    P : = nil
end PILEVIDE ;

procedure EMPILER (var P : PILE ; X : char ) ;
var E : PILE ;
begin  new (E) ; E ^. VAL : =X;
       E ^ . LIEN : = P ; P : = E
end EMPILER ;

procedure DEPILER (var P : PILE ) ;
begin if ESTVIDE (P)  then write ('erreur pile vide')
                      else P : = P ^ . LIEN
end DEPILER ;

function SOMMET (P : PILE ) : char ;
begin if ESTVIDE (P)  then write ('erreur pile vide')
                      else SOMMET:  = P ^ . VAL
end SOMMET ;

function ESTVIDE  (P : PILE ) : boolean ;
begin  ESTVIDE : = (P = nil )
end ESTVIDE ;
```

Les en-têtes de ces procédures sont les mêmes pour les deux représentations et correspondent à peu de choses près, à la signature du type pile.

Représenter un objet abstrait, c'est donc aussi savoir choisir parmi des représentations possibles, une ou des représentations efficaces dans le contexte du problème étudié. C'est pourquoi nous présentons dans le paragraphe suivant la notion de complexité d'un algorithme et plus particulièrement l'influence des choix de représentation de données sur l'efficacité d'un programme.

2 . 2 La notion d'efficacité en programmation

Faut-il se préoccuper d'efficacité ?

Avec l'avènement d'ordinateurs de plus en plus rapides, offrant de plus en plus d'espace mémoire, on peut se demander s'il est encore intéressant de chercher à améliorer l'efficacité des algorithmes et s'il n'est pas suffisant de s'attacher aux questions de lisibilité, de modularité, de correction. En fait, les applications traitées sont de plus en plus complexes et il existe des domaines "sensibles" comme la robotique ou l'aéronautique dans lesquels la prise de décision, le déclenchement d'opérateurs de secours par exemple, ne souffre aucun retard et appelle une rapidité maximum.

Il n'est donc pas suffisant de construire des programmes corrects et bien structurés, il faut aussi qu'ils soient efficaces. On rencontre plus généralement la notion de complexité d'algorithme ; nous allons donc définir la complexité d'un algorithme, indiquer comment on peut la mesurer pour nous attacher ensuite à l'une des caractéristiques d'un programme qui peut influer sur la complexité : la représentation de données.

La complexité d'un algorithme

Laurière rappelle dans [LAURI,86b] que "la complexité d'une procédure est la borne supérieure du nombre d'opérations qu'elle requiert, exprimée en fonction de la taille de l'énoncé d'entrée" et affirme à partir de cette définition que "la complexité d'un problème est la complexité du meilleur algorithme connu pour le résoudre", ce qui rend cette notion tout à fait dépendante de l'état de l'art !

Heureusement pour certaines classes de problèmes on sait jusqu'où on peut améliorer des algorithmes les résolvant.

Gaudel dans [GAU,86] parle de complexité en temps d'exécution et en espace mémoire : "La complexité d'un algorithme est le temps et l'espace mémoire nécessaires à son exécution".

Elle rappelle que si l'on veut étudier la complexité de programmes implantant des algorithmes, on fait des hypothèses sur les machines utilisées. La plupart des études concernent encore des machines séquentielles, mais l'avènement de machines parallèles apporte dans ce domaine des bouleversements sensibles. On suppose de plus les temps de transfert négligeables par rapport au temps d'exécution des opérations.

La mesure de la complexité

Il s'agit donc, en matière de complexité d'algorithmes, de quantifier deux grandeurs le "temps d'exécution" et la "place mémoire" utilisés dans le but de comparer entre eux différents algorithmes qui résolvent le même problème.

On peut être tenté de faire des mesures précises sur un ordinateur donné à partir

d'exécutions sur de nombreux jeux de données. L'analyse de telles mesures a conduit à des énoncés comme "l'algorithme A implanté par le programme P sur l'ordinateur O et exécuté sur la donnée D utilise K secondes de calcul et J bits de mémoire". En fait, le but des études de complexité d'algorithme est d'obtenir des résultats beaucoup plus généraux non liés à une machine.

Le temps d'exécution est évalué en fonction d'une grandeur du problème à résoudre, qui peut être la taille des données, des résultats, le rang ou la précision que l'on veut atteindre pour un calcul.

Si n représente la mesure de la taille, on dit que f(n) temps d'exécution de l'algorithme est de même ordre de grandeur (notation $O(n)$) qu'une fonction g(n) si :

il existe c et n_0 tels que $| f(n) | \leq c | g(n) |$ quelque soit $n > n_0$.

Les ordres de grandeur les plus fréquemment utilisés sont $O(1)$ (constant), n, log n, n log n, n^2, n^3. La place mémoire nécessaire est souvent évaluée en nombre d'octets, toujours en fonction de n.

Efficacité et représentation des informations

Une structuration adéquate des données est souvent un élément déterminant dans la conception d'algorithmes efficaces. Un exemple simple consiste à comparer le coût moyen de la recherche d'un élément dans une liste triée de n éléments (log n) et dans une liste non triée (n). Il concerne une opération particulière, la recherche d'un élément.

Beaucoup d'auteurs donnent l'exemple du monceau qui est un type spécial d'arborescence pouvant être implanté efficacement dans un tableau sans aucun problème. La propriété du monceau est que la valeur de chacun de ses noeuds internes est supérieure ou égale à celles de ses fils. C'est une excellent structure de données pour trouver le maximum, éliminer la racine, ajouter et modifier un noeud, c'est-à-dire implanter efficacement une liste de priorité dynamique. La valeur d'un noeud indique la priorité d'un évènement.

Cette représentation optimise ici un ensemble particulier d'opérations sur une information du problème.

Pour fournir au programmeur des critères de choix de représentation, nous allons donc associer à une opération ou à un groupe d'opérations, une mesure de complexité en temps d'exécution et en espace mémoire nécessaire pour chaque représentation connue.

Si le problème à résoudre n'utilise qu'une seule opération ou qu'un seul groupe d'opérations identifiées, le bon choix est facile à faire ! Mais dans la pratique, on applique plusieurs opérations à une information et une représentation peut optimiser l'exécution des unes tout en rendant inefficaces celle des autres ; le choix nécessite alors une analyse plus fine de l'algorithme et est souvent affaire de compromis.

2.3 REPRESENTATIONS ET CRITERES DE CHOIX DE REPRESENTATION

Pour chaque type, les auteurs proposent un certain nombre de représentations "standards" et parfois des représentations plus particulières comme le monceau que nous venons de décrire. Des critères de choix entre les différentes représentations proposées ne sont pas toujours explicitement exposés .

Exemples concernant les listes [GAU,86] :

"Nous donnons dans ce paragraphe différentes représentations des listes linéaires, des piles , des files et nous les comparons".

- Représentation contiguë dans un tableau

Appréciation proposée :

"...avec cette représentation, il est simple d'accéder au kième élément et de le parcourir séquentiellement. Il est aussi facile d'insérer ou de supprimer le dernier élément ; cependant lorsque l'élément à insérer ou supprimer est en position k < n, il faut déplacer tous les éléments de k à n dans le tableau pour reconstituer la structure".

- Représentation chaînée avec des pointeurs

Appréciation proposée :

"Cette représentation nécessite de la place mémoire supplémentaire pour les pointeurs, mais elle permet de traiter très rapidement la plupart des opérations sur les listes... cependant l'accès au kième élément n'est plus direct : on doit parcourir k-1 pointeurs à partir de la tête de la liste pour trouver le kième élément."

- Variantes de représentations avec pointeurs

Liste représentée par un pointeur sur un bloc de cellules et non sur une cellule

Appréciation proposée :

"L'utilisation d'un tel bloc permet d'éviter un traitement spécial pour l'insertion et la suppression en début de liste ".

Liste circulaire
"...utiles pour représenter des files"

Liste circulaire doublement chaînée
"...de nombreuses applications nécessitent de parcourir les listes à la fois vers l'avant et vers l'arrière".

Exemples de représentations concernant les ensembles

C. Pair propose dans [PAI,86] cinq représentations d'ensemble et étudie pour chacune d'entre elles le coût de certaines opérations et la place mémoire utilisée. Cette étude est résumée dans la figure suivante qui peut servir de "guide de choix" au programmeur.

"On peut résumer par le tableau suivant la complexité des opérations selon la représentation choisie pour un ensemble : les quatre premières lignes donnent l'ordre de grandeur du temps moyen nécessaire ; n est le nombre d'éléments de l'ensemble, v celui de V (si v est infini, la représentation par un tableau booléen est impossible).

	tableau booléen	liste chainée	liste contiguë	liste triée contiguë	arbre trié
appartenance, recherche	constant	n	n	log n	log n
ajout, suppr (en plus de la recherche)	constant	constant	n	n	log n
traitement de tous les éléments	v	n	n	n	log n
espace nécessaire	v	n	n	n	n

La représentation arborescente est donc un bon compromis entre la représentation contiguë et la représentation chaînée".

Certains auteurs vont plus loin en essayant de proposer un "graphe de décision" pour aider l'utilisateur dans ses choix. Voici, sur la page suivante, les résumés figurant dans [COU,84] pour les listes et les ensembles.

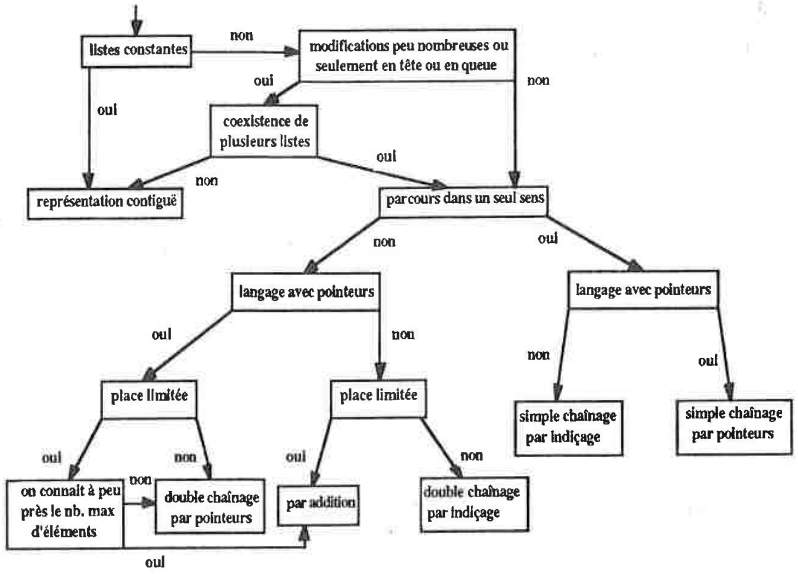


Diagramme d'aide au choix d'une implantation de liste

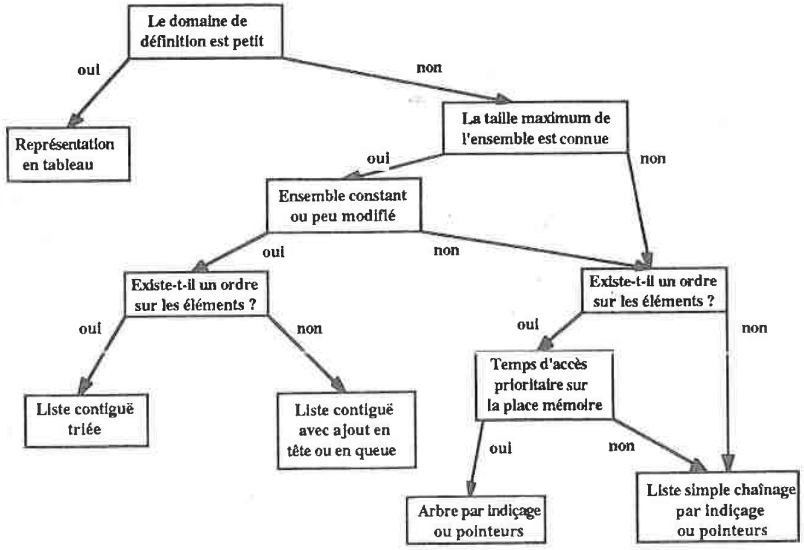


Diagramme d'aide au choix d'une implantation d' ensemble

2.4 CONCLUSION

Un des objectifs de cette rapide étude sur les types et leurs représentations était de rassembler la matière nécessaire à la constitution d'une base de connaissances sur les critères de choix d'une représentation efficace d'objet abstrait dans un contexte donné. Une première conclusion s'impose : nous avons trouvé des représentations différentes et pour chacune d'entre elles des critères permettant de dire qu'elle était meilleure qu'une autre dans certaines conditions d'utilisation. Dans la plupart des applications courantes, les conditions d'utilisation sont beaucoup plus complexes que celles énoncées et, pour ces applications, les critères de choix ne sont pas énoncés.

Une autre remarque concerne la prise en compte simultanée des différentes informations figurant dans une application ; il y est rarement fait allusion alors que c'est évidemment la situation la plus courante.

Chapitre 3

EXPRIMER DES ALGORITHMES PORTANT SUR DES TYPES ABSTRAITS DE DONNEES

La conception d'algorithmes utilisant des objets abstraits pose évidemment la question du mode d'expression de ces algorithmes. Dans un premier temps, chaque auteur a défini son langage [PAI,71], [AHO,74], [LIS,74], [COUV,84] et a indiqué comment passer manuellement d'un algorithme écrit dans ce langage à un programme dans le langage de programmation cible de l'utilisateur.

C'était un passage obligé et un mode de communication des algorithmes intéressant. Mais, comme nous le notions dès le début de cet ouvrage, il est difficile d'obtenir que cette étape de travail soit clairement identifiée tant que le langage support reste artificiel ; le programmeur est tenté de brûler des étapes et de travailler très vite dans le langage de programmation auquel il devra de toutes façons arriver.

Plusieurs auteurs ont donc réalisé une implantation de leur langage. Certains d'entre eux offrent ainsi un puissant environnement d'écriture de logiciels, qui, même s'il n'est pas diffusé dans l'industrie informatique, a permis de nombreuses réalisations dans le domaine de la recherche et de l'enseignement. CLU en est un bon exemple. D'autres ont simplement réalisé un précompilateur à usage pédagogique au dessus d'un langage comme PASCAL ; c'est le choix de [COUV,84] à Rennes.

Il existe aujourd'hui des langages de haut niveau offrant des primitives d'expression puissantes pour l'écriture d'algorithmes abstraits ; il nous paraît souhaitable de les utiliser de préférence à des notations exigeant ensuite une phase de traduction vers un langage cible. Nous tentons dans ce chapitre d'énumérer quelques-unes des caractéristiques que doit posséder un tel langage. Nous décrivons ensuite comment et jusqu'où ADA , le langage que nous avons choisi pour implanter notre système, remplit ce "cahier des charges" [LE

VER,82], [MAY,82], [DE MAS,82], [BOO,83], [BARN,84]. Nous évoquons enfin d'autres choix possibles.

3.1 CONCEPTS NECESSAIRES

Pour écrire des algorithmes abstraits, il faut disposer de mécanismes d'abstraction sur les données et sur les traitements. Les concepts fondamentaux sont la modularité, la structuration et la généricité.

3.1.1 La modularité

La modularité permet d'écrire des unités de programmes qui remplissent une fonction déterminée. Elle permet de regrouper soit des données, soit des traitements, soit les deux. Un module doit posséder des spécifications qui permettent de comprendre son rôle et de l'utiliser en faisant "abstraction" de détails de son contenu ; ces spécifications constituent sa partie visible et ses moyens de communiquer avec d'autres modules.

Une application est composée de plusieurs modules, il est donc essentiel d'en avoir une vue organisée qui peut être un schéma de décomposition de l'application, un graphe d'appel pour les modules de traitement ou d'utilisation pour les modules de définition de données.

3.1.2 La généricité

La généricité [JACQ,78], [BER,79] permet d'écrire des unités de programmes encore plus générales en passant en paramètres types de données et fragments de traitements.

Lorsqu'on passe des types en paramètre, il faut en outre pouvoir exprimer si n'importe quel type pourra être utilisé comme paramètre effectif ou bien si des conditions doivent être respectées. Meyer, dans [MEY,87a], parle de généricité non contrainte ou contrainte et donne l'exemple de deux éléments de type T dont on veut comparer les valeurs ; il est nécessaire qu' on ait défini et implanté un opérateur de comparaison sur le type effectif.

Lorsqu'on passe des procédures ou fonctions en paramètres, la question est de savoir si le profil de telles procédures (nombre de paramètres, types des paramètres et du résultat) peut également être paramétré ou non.

La puissance d'expression d'un tel langage vient aussi du nombre de primitives de

haut niveau qu'il offre à l'utilisateur. Comme un langage avec un très grand nombre de primitives de haut niveau adaptées à des domaines d'application variés est irréaliste, il faut que les mécanismes implantant les notions de modularité et de généricité permettent à l'utilisateur de créer lui-même les primitives dont il a besoin. Pour les données, il faut pouvoir définir des types et des constructeurs de types.

Si le langage d'expression des algorithmes abstraits doit aussi être le langage d'expression de programmes efficaces, il faut pouvoir associer plusieurs représentations à un objet abstrait et aux opérations effectuées sur des objets de ce type, puis disposer si possible d'un mécanisme de choix de la représentation la mieux adaptée au problème.

D'autres qualités peuvent être recherchées, comme la possibilité d'exprimer facilement des modifications de spécifications, de réutiliser des composants existants, etc... Mais nous nous en tenons aux principales que nous venons d'évoquer et nous étudions maintenant comment elles se traduisent dans le langage ADA et avec quelles limites.

3.2 UN CHOIX POSSIBLE : ADA

3.2.1 Le langage ADA (J. Ichbiah, 1979)[ICH,84], [ICH,85]

Les origines du langage

Le langage ADA est né entre 1975 et 1979 de la volonté du Ministère américain de la défense (DoD). Le DoD souhaitait faire baisser le coût de production du logiciel en disposant d'un langage :

- bien adapté aux contraintes des applications "embarquées" pour lesquelles il développait de nombreux logiciels, c'est à dire permettant notamment des traitements en parallèle, du contrôle en temps réel, du traitement d'exceptions et une gestion uniforme des entrées-sorties,

- favorisant l'utilisation des concepts de programmation moderne, notamment celles de modularité, de structuration et d'abstraction.

Il faut donc se souvenir que, même si l'utilisation de ADA à l'échelle industrielle est récente, la modernité des idées incarnées dans ce langage est celle des années 75.

Objectifs du cahier des charges

ADA est un langage destiné à favoriser la construction de programmes par affinements successifs, il doit donc permettre de couvrir toutes les phases du processus de construction, d'un niveau élevé de description d'algorithmes jusqu'aux détails d'implantation efficace et aux modifications ultérieures. On trouve dans le cahier des charges les caractères suivants :

- des constructions facilitant la programmation structurée
- un typage fort
- des spécifications relatives ou absolues quant à la précision des calculs
- des mécanismes de visibilité et d'abstraction des données
- du traitement parallèle
- la prise en compte d'exceptions
- des définitions génériques
- la possibilité de définir des caractéristiques dépendant d'une machine.

Nous intéressons dans la suite de la présentation aux mécanismes offerts pour prendre en compte la description de données abstraites indépendamment d'une implantation particulière.

Il faut noter que les environnements intégrés spécifiés dans le cahier des charges ont été livrés avec du retard par rapport aux composants indispensables et qu'ils commencent seulement à être utilisés [Revue Génie Logiciel, n° 1]. Mais l'idée de bibliothèques de programmes réutilisables est à penser dans ce contexte.

3.2.2 Les mécanismes d'abstraction en ADA

Un programme ADA

Un programme ADA se compose d'une ou plusieurs unités qui peuvent être compilées séparément. Une unité de programme est un sous-programme, une tâche (task) ou un paquetage (package).

Une tâche définit une action qui peut logiquement être exécutée en parallèle avec d'autres. C'est le mécanisme d'expression du parallélisme offert en ADA ; il ne concerne pas notre application.

Un sous-programme est soit une procédure, soit une fonction.

Un paquetage est un ensemble de ressources (type de données, objets sous-programmes, tâches, autres paquetages) que l'on veut regrouper.

Chaque unité comprend deux parties, une spécification et un corps (body). La spécification contient les informations utiles à l'utilisateur et visibles par lui. Le corps contient les détails d'implantation qui peuvent être cachés à l'utilisateur.

Des exemples de textes de programmes ADA ont été fournis dans le premier chapitre de la partie I. Nous allons donc nous intéresser directement aux mécanismes d'abstraction de ADA, sachant qu'au point de vue syntaxe et structures de contrôle usuelles, les notations ne devraient pas dérouter un lecteur familier de PASCAL.

La modularité

ADA offre essentiellement deux constructions syntaxiques pour permettre une programmation modulaire, les sous-programmes et les paquetages.

- Les sous-programmes

Exemples:

```
function   HACHAGE (CLE : in ELEMENT) return ADRESSE is
begin
.......
liste d'instructions
.......
end HACHAGE;


procedure COMPTE_MOTS is
begin
.......
liste d'instructions
.......
end COMPTE_MOTS;
```

Le début du texte (jusqu'au symbole **is**) constitue la spécification du sous-programme ; in, out, in out indiquent les fonctions des paramètres entrée, sortie, entrée/sortie.

- Les paquetages

Exemple :

```
package COMPLEXE is
  type NOMBRE is record
      PARTIE_REELLE : FLOAT ;
      PARTIE_IMAGINAIRE : FLOAT ;
              end record ;
```

```
function "+" (A, B : in NOMBRE) return NOMBRE ;
function " -" (A, B : in NOMBRE) return NOMBRE ;
function " *"(A, B : in NOMBRE) return NOMBRE ;

end COMPLEXE ;
```

Ce texte constitue la spécification d'un paquetage pour calculer sur des nombres complexes. Les opérateurs +, -, * y sont redéfinis comme fonctions et peuvent être utilisés sur des objets du type COMPLEXE avec la signification qui est précisée dans le corps de paquetage. Il doit être accompagné d'un corps de paquetage, non nécessairement visible de l'utilisateur, dans lequel sont définis les corps des fonctions.

```
PACKAGE BODY COMPLEXE is
.....
textes des fonctions
.....
end COMPLEXE ;
```

La notion de visibilité

Globalement en ADA, la notion de visibilité est analogue à celle d'autres langages ; seule est visible la partie spécification des différentes unités. Toutefois, dans les paquetages, il est possible de définir des types "private " ou "limited private" dont nous expliquons le rôle.

Exemple de partie "private" de la spécification d'un type ensemble (I° partie, chapitre 1) :

```
private
type TABV is array  (Natural range <>) of ELEM ;
type ENS (T_MAX : TRANG) is
        record
            TAILLE : TRANG ;
            TVAL : TABV (0..T_MAX) ;
        end record ;
```

Ce texte décrit la représentation retenue pour un objet de type ensemble représenté comme un tableau de valeurs dans le paquetage créé. Cette représentation n'est utilisable que dans les corps de procédure implantant les opérations du type. Elle est "privée" pour ce paquetage. Les objets du type ENSEMBLE ainsi représenté doivent être manipulés avec les opérations du type, on peut également effectuer sur eux des affectations et comparaisons.

Si l'on veut interdire ces dernières opérations, on peut déclarer les types en "limited private".

La généricité

La notion de généricité augmente encore la puissance d'expression des autres concepts mis en oeuvre en ADA. Elle permet de réaliser des sous-programmes et paquetages capables de manipuler des objets dont le type est donné en paramètre lors de la spécification et d'utiliser des procédures dont le nom est également fourni en paramètre, comme l'illustre l'exemple suivant :

```
generic
    type ELEM is private ;
    type TAB is array (INTEGER range <>) of ELEM ;

    with function "<" (X, Y : in ELEM) return BOOLEAN is <> ;
    procedure TRI (T : in out TAB) is
        corps de procédure de tri
    end TRI ;
```

La procédure TRI ci-dessus est doublement générique, elle utilise d'une part deux types paramétrés pour les éléments du tableau et le tableau à trier, d'autre part une fonction de comparaison notée "<". Pour l'utiliser, il faut créer un exemplaire particulier de cette procédure en substituant deux types et une fonction aux paramètres génériques de cette déclaration ; ce qui s'écrirait par exemple :

```
...
    type PRODUIT is
      record
        CODE_ARTICLE : INTEGER range 1... 1000 ;
        PRIX                 : REAL ;
      end record ;

    type TAB_PRODUIT is
        array (INTEGER range < > ) of PRODUIT ;

    MES_PRODUITS : TAB_PRODUIT ;

    function  COMPAR_PRIX (X, Y : in PRODUIT ) return BOOLEAN is
        begin
        if X.PRIX ≤ Y.PRIX then return  (TRUE) ;
                                else return  (FALSE) ;
        end if ;
    end COMPAR_PRIX ;

procedure TRIER_PRODUITS_PRIX is new
    TRI (ELEM =>PRODUIT, TAB =>TAB_PRODUIT, "<" => COMPAR_PRIX) ;
```

```
-- corps de procédure principale.
    begin
    ----
    TRI ( MES_PRODUITS) ;
    end ;
```

On peut donc dans un paquetage paramétrer des types utilisés dans le paquetage ainsi que certaines procédures ; c'est de cette façon que l'on réalise des spécifications de types abstraits de données.

3.2.3 Spécifications et implantations d'un type abstrait de données en ADA

Le programmeur peut en ADA créer ses propres types et constructeurs de type en utilisant les paquetages, les types privés et la généricité.

La partie spécification du paquetage contient les paramètres génériques du type, les profils des opérations définies sur ce type, une description de représentation des objets du type. La partie corps de paquetage contient les programmes réalisant l'implantation des opérations du type.

Nous donnons au chapitre 1 de la troisième partie un paquetage complet pour le type LISTE, comme illustration du contenu de la bibliothèque de types.

3.2.4 Quelques limites à l'utilisation de ADA

Nous signalons dans ce paragraphe quelques types de difficultés rencontrées dans l'utilisation de ADA comme langage d'expression d'algorithme sur des types abstraits de données et d'implantations d'objets de ces types.

Difficultés liées à la notion de paquetage

Un paquetage décrivant un type abstrait comprend une partie spécification et une partie implantation. Lorsqu'on veut fournir différentes implantations pour un même type, il faut écrire autant de paquetages que d'implantations alors que toutes les parties spécifications (à l'exception de la définition de la représentation, partie "private") sont identiques.

Difficultés liées aux procédures et fonctions génériques

L'expression suivante est excessivement lourde :


```
generic
with function f(... ) ;
procedure truc (     ) is
        ..................
end truc ;

function ma_f(...  ) is
        ..................
end ma_f ;

procedure mon_truc is new truc (f => ma_f) ;
...
-- utilisation de mon_truc --
```

Or, on en a besoin pour certaines des opérations relatives à un type.

Exemple : PARCOURS de chaque élément d'une liste paramétré par une procédure ACTION contenant la description du traitement à appliquer à chaque élément.

Le profil de ACTION est entièrement déterminé à la spécification et ne permet donc pas d'instancier:

un parcours1dans lequel ACTION 1 aurait 3 paramètres,

un parcours2 dans lequel ACTION 2 aurait 2 paramètres.

Enfin, si l'on est amené à instancier deux procédures parcours provenant de deux paquetages différents, l'ambiguïté de noms ne peut pas être levée, et il faut préfixer le nom de procédure par le nom de type.

Et pour conclure, ce mode d'écriture ne permet pas de retrouver exactement les noms d'opérations du type dans le corps de l'algorithme.

Difficultés liées aux fichiers

Certains caractères types de fichiers (descripteurs) proviennent d'autres paquetages et empêchent de déclarer "private" les types correspondants dans les paquetages... Les "types abstraits" ainsi implantés ne peuvent avoir la même forme que les autres dans leur partie spécification.

3.3 D'AUTRES SOLUTIONS

Modula 2 construit par Wirth [WIR,84] au dessus de Pascal offre aussi le concept de module, il admet le type "procédure" ce qui permet notamment de passer des descriptions de

traitements en paramètre et de leur affecter des valeurs dans les programmes. Mais il ne possède pas la généricité de type dans les modules.

CLU offre un bon environnement d'écriture d'algorithmes abstraits et d'implantation de ces algorithmes.

On voit enfin apparaître des langages plus riches, c'est à dire offrant à la fois typage et généricité et héritage comme EIFFEL [MEY,87a&b]. Avec des langages de ce type, il faut sans doute repenser aussi la modélisation de l'univers pour utiliser toute leur puissance.

CONCEPTION D'UN SYSTEME D'ENSEIGNEMENT SUR LA REPRESENTATION DE TYPES ABSTRAITS DE DONNEES

III° Partie

Introduction à la partie III

Après avoir présenté le problème de l'utilisation de types abstraits et de leur implantation dans les langages algorithmiques usuels, ses contextes, les manuels d'enseignement relatifs à cette question, nous consacrons cette troisième partie à la conception d'un système d'enseignement destiné à accompagner un cours sur la représention des types abstraits de données.

Faire des choix d'implantation pour les objets abstraits, c'est mettre en oeuvre une expertise dans le domaine de la construction de programmes. Nous avons vu dans les deux parties précédentes qu'une telle expertise était loin d'être bien formalisée dans les manuels et traités consacrés à la question, mais que des recherches actuelles, tentant d'automatiser certaines phases du processus de construction de programmes, essayaient d'expliciter une part des connaissances nécessaires.

Un premier objectif de cette troisième partie est donc d'exprimer une telle expertise ; nous le faisons dans le chapitre 2 en construisant des bases de connaissances contenant des critères de choix d'implantations pour des objets de type LISTE et ENSEMBLE. Nous montrons comment un système qui les utilise permet de résoudre par exemple certains exercices étudiés dans la première partie.

Une fois cette expertise exprimée, nous nous intéressons à la transmettre à des élèves. Le chapitre 3 est donc consacré à présenter différentes utilisations pédagogiques des systèmes à bases de connaissances et à décrire les solutions que nous avons retenues pour notre système. Ce système n'est pas conçu pour être l'unique instrument d'apprentissage des élèves dans ce domaine ; il s'insère au contraire dans un processus global d'enseignement et nous proposons donc plusieurs modes d'utilisation entre lesquels un enseignant devra choisir en fonction de son projet pédagogique.

Pour apprendre à l'élève à passer d'un algorithme abstrait à un programme efficace,

il lui faut un guide de raisonnement mais aussi un environnement favorisant une telle démarche. Nous avons choisi ADA parce qu'on peut y exprimer à la fois des procédures portant sur des objets abstraits et les programmes implantant ces objets et les opérations effectuées sur eux. La bibliothèque de types et de représentations mise à disposition de l'élève dans le système est décrite au chapitre 1.

Nous avons réalisé ce système tout en observant notre activité de conception afin que la construction de cette application puisse constituer une étude de cas montrant la mise en oeuvre de méthodes et d'outils, expliquant les choix opérés et les difficultés rencontrées, comparant les solutions retenues avec les démarches suggérées dans d'autres domaines.

Ces chapitres contiennent donc à la fois la description de la base de connaissances réalisée et, par étapes, des réflexions sur les modalités de création de cette base qui sont résumées à la fin du chapitre 2 en ce qui concerne la structuration de l'univers de travail et des raisonnements nécessaires à la résolution des problèmes étudiés et en fin de chapitre 3 pour ce qui a trait aux utilisations pédagogiques de la base de connaissances.

Nous concluons sur les améliorations qui pourraient, sur divers plans, être apportées à un tel système.

Chapitre 1

LA BIBLIOTHEQUE DE SPECIFICATIONS ET D'IMPLANTATIONS DE TYPES

Pour réaliser un environnement d'enseignement permettant à un élève de mettre en oeuvre les notions de programmation abstraite et d'implantations efficaces des objets de l'univers de travail, nous faisons le choix de donner une bibliothèque de spécifications et d'implantations de types de données en ADA. La base de connaissances sur les critères de choix de représentation d'un objet sera conçue pour être utilisée avec cette bibliothèque.

Nous avons déjà donné deux des raisons qui peuvent justifier ce choix dans les chapitres précédents :

- Nous voulons utiliser des outils de production de logiciel largement diffusés chez les professionnels ; ADA bénéficie d'une norme internationale, il est en cours de diffusion ; les besoins de formation aux concepts qui font l'objet de notre étude sont grands et peuvent justifier un développement de logiciels pour l'enseignement.

- Ce langage nous permet de satisfaire une part importante des caractères et propriétés qui nous ont semblé nécessaires à l'issue de notre étude sur l'enseignement des types de données et de leurs représentations [FEL,87].

De plus, une telle bibliothèque présente d'autres intérêts :

Les utilisateurs du système vont apprendre à ne pas écrire ex nihilo toutes les lignes de programme nécessaires à la confection d'un logiciel, mais plutôt à travailler dans un environnement existant, ce qui est plus conforme à la situation quotidienne de l'informaticien professionnel. Cependant, la programmation par réutilisation de composants qu'ils vont ainsi pratiquer présente des avantages et des inconvénients par rapport à la construction de programmes par affinements successifs qu'il convient de signaler ; c'est l'objet du premier paragraphe. Enfin, la bibliothèque fournit des textes de procédures ADA où un effort de

structuration et de cohérence dans la présentation a été fait avec l'espoir que naissent de leur lecture de bonnes habitudes d'écriture de programmes comme le suggèrent Hibbard et ses co-auteurs dans [HIB,81].

Nous présentons au paragraphe 2 les types prévus dans la bibliothèque et nous expliquons les raisons qui nous ont amenés à choisir les opérations et implantations retenues. Nous montrons au paragraphe 3 sous quelle forme figurent dans la bibliothèque les informations et les modules ADA relatifs à un type et illustrons notre propos par des exemples de spécifications et d'implantations commentés.

1.1 LA PROGRAMMATION PAR REUTILISATION DE COMPOSANTS

1.1.1 Un mode de travail souvent utilisé dans l'industrie

O. Lecarme et R. Rousseau [LEC,85] terminent leur fascicule sur les langages de programmation, types et modularité, par un paragraphe sur les composants logiciels standards:

"En regardant de plus près les composants d'une application nouvelle ou prétendue telle, on constate que bien souvent celle-ci est en fait une articulation particulière de sous-problèmes classiques. Cela laisse augurer que la programmation qui exploitera intensément les possibilités génériques ressemblera davantage aux autres domaines de fabrication industrielle où l'on effectue du montage de composants standards. Certains fabriquants de logiciels vendront des composants logiciels comme cela se fait pour les engrenages ou les composants électroniques. La programmation aura gagné en efficacité."

1.1.2 Avantages et inconvénients

Ce mode de travail permet la construction d'un véritable environnement de programmation qui peut à tout moment être enrichi ou modifié et surtout adapté aux besoins des utilisateurs. Les programmes produits bénéficient de la fiabilité des composants utilisés. Le programmeur gagne du temps dans l'écriture de son application, il n'hésite pas à optimiser une section critique du code en recherchant un composant plus efficace.

L'inflation du nombre de composants nécessite une assistance à l'utilisateur pour parcourir une bibliothèque très fournie. La standardisation ainsi opérée peut exclure en première approche certaines solutions originales que l'imagination du programmeur expérimenté proposerait devant une situation donnée... On retrouve là le problème plus général de la fabrication industrielle par rapport à la création artisanale ! Le programme

produit peut perdre en efficacité, notamment lorsque le composant refait une part de travail déjà accompli par ailleurs, par exemple pour des contrôles.

Nous en restons volontairement à des remarques de nature générale ; il faudrait étudier de façon plus précise cette notion de composant en programmation et noter par exemple que le grand nombre de composants nécessaires est dû notamment à des "manques" au niveau de la mise en oeuvre du concept de généricité qui obligent à créer des composants différents pour de faibles modifications de composants existants.

1.1.3 ADA et la réutilisation de composants

Cette notion de "composant logiciel" est très présente dans beaucoup d'ouvrages qui traitent aujourd'hui des applications du langage ADA ; elle est même le thème central de plusieurs d'entre eux :

Booch [BOO,87] a publié "Software Components with ADA " (600 pages) avec l'objectif affiché de fournir à la communauté informatique des composants dans le domaine des types de données et d'algorithmes utilitaires comme des tris.

Bardin et Thompson insistent dans un article [BARD,88] intitulé "Composants logiciels ADA composables et le paradigme d'exportation", sur le fait que l'un des buts de la communauté internationale de l'ingénierie du logiciel est de créer une industrie de composants logiciels réutilisables. Il note comme une difficulté possible dans ce développement le manque de qualités de "composition" entre composants permettant de créer de nouvelles abstractions à partir d'autres existantes.

Dans la réalisation de gros projets de génie logiciel, les outils mis à la disposition du programmeur sont souvent plus importants que le langage dans lequel l'application est codée. Conscient de cette réalité, le DoD avait fait définir, en même temps qu'un cahier des charges de langage, les caractéristiques d'un environnement de développement de logiciels dans ce langage nommé APSE (ADA Programming Support Environment) dans les documents de présentation du langage.

L'idée est d'implanter un tel environnement sur une machine et de le faire par couches successives selon les caractères des machines et des applications envisagées. Au dessus du système d'exploitation de la machine, se trouve un premier noyau qui sert d'interface, le KAPSE (Kernel ADA Support Environment), puis le MAPSE (Minimal ADA Programming Support Environment). Au delà, viennent les composants propres à l'utilisateur et à son domaine de travail ; c'est donc à ce niveau que vient se placer la bibliothèque dont nous décrivons maintenant le contenu.

Il faudrait par ailleurs enseigner la création de composants logiciels comme le suggère J.P. Rosen dans [ROS,86], mais c'est une autre question que nous ne traitons pas ici.

1.2 ENVIRONNEMENT RETENU

1.2.1 Les types proposés

L'analyse du contenu à enseigner nous a amenés à retenir les types génériques les plus fréquemment présentés, à savoir :

LISTE avec ses cas particuliers PILE et FILE
ENSEMBLE
TABLE
ARBRE
GRAPHE.

Cette liste peut paraître limitée par rapport au choix de certaines publications ; nous pensons qu'elle est suffisante dans un environnement d'enseignement pour rendre compte de la diversité des problèmes rencontrés dans le domaine et de solutions variées pour les résoudre. Nous ne construisons pas un environnement de production en vraie grandeur.

Pour chacun de ces types, il faut choisir la définition précise que l'on veut en donner, le jeu d'opérations qui seront implantées ainsi que les représentations proposées.

1.2.2 Les spécifications d'un type

Chacun des types retenus évoque en fait un ensemble de variantes proches mais non identiques. Le choix d'un exemplaire parmi les variantes possibles est indiqué :

- de manière informelle dans un paragraphe de présentation intuitive du type,
- de manière formelle par la donnée de la signature du type et des axiomes associés.

Ce choix fixe aussi l'ensemble des opérateurs disponibles pour ce type. Certains opérateurs, les générateurs et quelques autres qui caractérisent le type comme le test d'appartenance d'un élément à un ensemble, figurent nécessairement dans la définition. Pour d'autres, la décision de les retenir ou de les éliminer n'est pas évidente à prendre.

Si l'on choisit un nombre minimal d'opérations, on fournit un environnement que l'utilisateur maîtrisera plus facilement et on lui laisse définir et représenter lui-même celles qui

lui sont nécessaires et qui ne figurent pas dans la définition. Cela peut permettre des solutions mieux adaptées aux problèmes mais alourdit quelque peu le travail de programmation.

Si, au contraire, on fournit un grand nombre d'opérations, ce qui semble être la tendance d'environnements comme CLU [LIS,86] avec 22 opérations pour le type "séquence" ou Software Components with ADA [BOO,87], on rend la maîtrise du jeu d'opérations plus difficile pour le débutant, on offre un confort de travail plus important au développeur déjà expérimenté. Un autre argument qui plaide en faveur d'un grand nombre d'opérateurs est celui de l'identification de certaines opérations particulières qui justifient des implantations adaptées ; ces opérations pourront être identifiées lors d'une analyse statique du texte de l'algorithme.

Nous avons essayé de trouver un juste milieu entre ces deux solutions extrêmes et nous nous sommes attachés à bien structurer la liste des opérations mettant en évidence le rôle de chacune d'entre elles et son importance relative par le rang qu'elle occupe dans la description.

1.2.3 Les implantations proposées

Nous avons déja évoqué au premier paragraphe l'incidence du nombre de représentations proposées pour un type donné sur la taille de la bibliothèque : il faut créer un paquetage par représentation. Nous avons sur cette question aussi, adopté une solution médiane en proposant les représentations les plus classiques.

<u>Implantations proposées pour le type LISTE</u>

Implantations en mémoire centrale : les objets de type LISTE sont représentés à l'aide du constructeur de type article (record) avec les champs indiqués ci-dessous :

Implantation contiguë : (taille, tête, queue, tableau de valeurs)
Implantations chaînées :
simple chaînage (taille, tête, queue, pointeur d'accès à élément (val, suiv))
double chaînage (taille, tête, queue, pointeur d'accès à élément (val, suiv, pred))

Implantations en mémoire externe : des objets de type LISTE sont définis en utilisant les paquetages prédéfinis SEQUENTIAL_IO pour une implantation en fichier séquentiel et DIRECT_IO pour une implantation en fichier à accès direct.

Remarques

Toutes les opérations définies dans la spécification ne peuvent pas être implantées pour toutes les représentations ; cela est vrai notamment pour les représentations de listes dans des fichiers. Dans certains cas, l'opération peut être implantée mais elle ne peut avoir le profil standard.

Exemple : L'adjonction d'un élément dans une liste a pour nom standard ADJL et pour profil

ADJL (IDLIST : in out LISTE ; RANG : in TRANG ; IDVAL : in ELEM).

Dans le cas d'une représentation de liste par un fichier séquentiel, ce profil ne peut être utilisé car on ne peut insérer un élément dans un fichier ; il est nécessaire de créer un nouveau fichier et de fournir une opération ADJ avec le profil

ADJ(IDLIST: in LIST; RANG: in TRANG; IDVAL: in ELEM; IDLIST_SORT: in out LIST)

Dans d'autres cas, nous avons choisi de ne pas proposer d'implantation, comme par exemple pour l'opération PRED dans un un fichier séquentiel. Nous indiquons simplement en commentaire de la spécification que l'implantation correspondante n'est pas disponible.

Implantations proposées pour le type ENSEMBLE

Implantations contiguës :

En mémoire externe : fichier à accès direct.

En mémoire centrale : tableaux.

- tableau de booléens indexé par le type des éléments du référentiel

Exemple : L'ensemble de caractères E = {A, E, I, O, U} est représenté par un tableau T de 26 éléments de type booléen, dans lequel T[CAR] est vrai si CAR appartient à E et faux sinon.

- tableaux de booléens avec "table majeure"

Exemple : Pour représenter des ensembles de mots appartenant tous à un même référentiel, on construit une table permettant d'associer à chaque mot du référentiel son rang dans la liste ordonnée de ses mots, puis on représente chaque ensemble de mots par un tableau de booléens indexé sur un intervalle d'entiers. L'accès à un élément se fait par recherche dichotomique du rang de l'élément dans la table associée au référentiel, puis accès à la place de l'élément dans le tableau représentant l'ensemble.

- tableau non trié des éléments de l'ensemble

- tableau trié des éléments de l'ensemble

Implantations chaînées entre elles par pointeurs :

- chaîné non trié
- chaîné trié
- arbre binaire

Technique mixte :

Une fonction de hachage permet de répartir les éléments de l'ensemble en classes, dans chaque classe on chaîne entre eux les éléments de la classe.

1.3 LE CONTENU DE LA BIBLIOTHEQUE POUR UN TYPE

Pour chaque type, la bibliothèque contient un texte de présentation du type et autant de paquetages ADA qu'il est prévu d'implantations différentes pour ce type.

1.3.1 Exemple de présentation :

```
NOTION REPRESENTEE :
    LISTE

DEFINITION
    Une liste est un objet caracterise par la notion de
    parcours sequentiel auquel est associe la notion de rang
    qui definit une relation d'ordre sur les elements de la
    liste.


DEFINITION SYNTAXIQUE
    Specification algebrique

    Sorte : Liste de ELEM

    Utilise : element de type ELEM

    Les operations :

        Generateurs
        Creer_list : -> LISTE
        Adjl : LISTE, TRANG, ELEM -> LISTE
        Ajout_apres : LISTE, PLACE, ELEM -> LISTE

        Observateurs
        Taille : LISTE -> ENTIER
        Est_vide : LISTE -> boolean
        Tete : LISTE -> PLACE
        Queue : LISTE -> PLACE
        Acces : LISTE, TRANG -> PLACE
        Succ : LISTE, PLACE -> PLACE
        Pred : LISTE, PLACE -> PLACE
        Valeur : LISTE, PLACE -> ELEM

        Modificateurs
        Supl : LISTE, TRANG -> LISTE
        Chglp : LISTE, PLACE, ELEM -> LISTE

        Iterateurs
        Init_list(saisie) : LISTE -> LISTE
        Parc_total(fonc) : LISTE -> LISTE
        Prem_tel_que(predicat) : LISTE -> RANG, PLACE
        Parc_partiel(predicat,fonc):LISTE,PLACE->LISTE,PLACE

        Divers
        Copie : LISTE -> LISTE
        Concat : LISTE, LISTE -> LISTE
```

```
Implantations definies

lis_manager_ctg
      implantation par un tableau en memoire centrale
      la taille du tableau est fixee par l'utilisateur

lis_manager_chn
      implantation par chainage simple en memoire centrale

lis_manager_chd
      implantation par chainage double en memoire centrale

lis_manager_fil
      implantation par fichier sequentiel
        : fonctions et procedures suivantes changent de
    profil
          parc_total(fonc)
          parc_partiel(fonc)
          prem_tel_que(predicat)
          copie
          concat
     les fonctions et procedures suivantes changent de nom
    et de profil
          adjl devient adj
          supl devient sup
          chglp devient chgp
          ajout_apres devient ajout
     les fonctions et procedures suivantes ne sont pas
    definies pour cette implantation
          tete, queue, succ, pred
     les fonctions et procedures suivantes sont
    particulieres a cette implantation
          creer_la_liste_a_lire

lis_manager_fid
      implantation par fichier a acces direct
      les fonctions et procedures suivantes changent de
    profil
          copie
          concat
     les fonctions et procedures suivantes sont
    particulieres a cette implantation
```

1.3.2 Un exemple complet de paquetage :

Nous avons vu au chapitre 3 qu'un paquetage est un module de programme ADA qui peut être utilisé pour implanter la notion de type abstrait de données ; il contient toutes les informations relatives au type de donnée traité. Il se compose en deux parties :

- Une partie spécification indiquant le nom du ou des éléments génériques qu'il faudra remplacer par des noms de types ou de prodédures effectifs lors de l'utilisation du paquetage, les noms et profils des opérations utilisables sur ce type. Cette partie contient les seules informations utilisables par le programmeur.

- Une partie implantation des objets du type qui contient le texte des procédures implantant les opérations définies précédemment ; du point de vue de la "visibilité" dans les programmes, l'utilisateur n'a pas besoin de les connaitre pour travailler.

Les pages suivantes contiennent à titre d'exemple le paquetage de la représentation contiguë "tableau de valeurs" du type LISTE proposée dans le système.


```
        -- ------------------------------------------------   --
        --       fichier : liste_spec.a

        --      DEFINITION D'UN INTERFACE (PACKAGE) POUR LA     --
        --    MANIPULATION DES OBJETS DU TYPE ABSTRAIT "LISTE"  --

        -- ------------------------------------------------   --

generic
      Type ELEM is private;

package LIST_MANAGER is

      Subtype TRANG is INTEGER range 0 .. 10;
      Subtype PLACE is INTEGER range 0 .. 10;
      Type LISTE is private;

-- ------------------------------------------------------------------
--             Declarations de procedures
-- ------------------------------------------------------------------
      -- Generateurs ---------------------------------------------
      Procedure CREER_LIST(IDLIST : in out LISTE);

      Procedure ADJL          (IDLIST : in out LISTE;
                               RANG   : in TRANG;
                               IDVAL  : in ELEM);

      Procedure AJOUT_APRES   (IDLIST : in out LISTE;
                               PL     : in PLACE;
                               IDVAL  : in ELEM);

      -- Observateurs ---------------------------------------
      Function TAILLE (IDLIST : in LISTE) return INTEGER;

      Function EST_VIDE       (IDLIST : in LISTE ) return BOOLEAN ;

      Function TETE   (IDLIST : in LISTE) return PLACE;

      Function QUEUE  (IDLIST : in LISTE) return PLACE;

      Function ACCES  (IDLIST : in LISTE;
                       RANG   : in TRANG) return PLACE ;

      Function SUCC   (IDLIST :in LISTE;
                       PL     : in PLACE) return PLACE;

      Function PRED   (IDLIST :in LISTE;
                       PL     : in PLACE) return PLACE;

      Function VALEUR (IDLIST : in LISTE;
                               PL     : in PLACE) return ELEM;

      -- Modificateurs -------------------------------------------------
      Procedure SUPL          (IDLIST : in out LISTE;
                               RANG   : in TRANG);

      Procedure CHGLP          (IDLIST : in  out LISTE;
                               PL     : in  PLACE;
                               IDVAL  : in ELEM);

      -- Autres operateurs ---------------------------------------
      generic
      with procedure SAISIE (e : in out ELEM ; encore : in out BOOLEAN);
      Procedure INIT LIST (IDLIST     : in out LISTE);
```

```
        generic
        with procedure FONC(E: in  out ELEM );
        Procedure PARC_TOTAL (IDLIST: in out LISTE; PL :out PLACE);

        generic
        with function PREDICAT(E : in ELEM) return BOOLEAN;
        Procedure PREM_TEL_QUE  (IDLIST : in LISTE;
                                 RANG   : out TRANG;
                                 PL     : out PLACE);

        generic
        with procedure FONC(E : in out ELEM );
        with function PREDICAT(E : in  ELEM) return BOOLEAN;
        Procedure PARC_PARTIEL (IDLIST  : in out LISTE;
                                DEB     : in PLACE ;
                                FIN     : out PLACE);

        Procedure COPIE (IDLIST : in LISTE;
                         IDLIST_SORT    : out LISTE);

        Procedure CONCAT        (IDLIST_DEB     : in LISTE;
                                 IDLIST_SUIT :   in LISTE;
                                 IDLIST  :       out LISTE);

        -- Definition de la representation ----------------------------------
        private
        Type TABV is array (Natural range <> ) of ELEM;
        Type LISTE is
                record  T: TRANG;       -- taille de la liste
                        TETE : PLACE ;
                        QUEUE : PLACE;
                        TVAL    : TABV(0 ..10);
                end record;

end LIST_MANAGER;
```



```
        -- ------------------------------------------------   --
        --       fichier : lis_manager_ctg_body.a

        --      DEFINITION D'UN INTERFACE (PACKAGE) POUR LA     --
        --    MANIPULATION DES OBJETS DU TYPE ABSTRAIT "LISTE"  --

        -- implantation par tableau
        -- d'elements de type entier, reel, caractere, chaine (?)
        -- ------------------------------------------------   --
with text_io; use text_io;

package body LIS_MANAGER_CTG is

 -- ---------------------------------------------------------------------
 --              Definitions des procedures
 -- ---------------------------------------------------------------------


Procedure CREER_LIST(IDLIST : in out LISTE) is
        begin
        idlist.t := 0;
        idlist.tete := 0;
        idlist.queue := 0;
        end CREER_LIST;

Procedure INIT_LIST (IDLIST: in out LISTE) is
        val : ELEM;
        encore : boolean := true ;
        PL : PLACE;
        begin
        pl := 1;
        while encore loop
                SAISIE(val, encore);
                IDLIST.tval(pl):= val;
                IDLIST.t := idlist.t+1;
                pl := pl+1;
                if IDLIST.TVAL'LAST = pl then raise SATURATION; end if;
        end loop;

        exception
        when SATURATION =>
                put("La liste est pleine; cf declaration et taille");
        end INIT_LIST;

Procedure ADJL(IDLIST: in out LISTE;RANG: in TRANG;IDVAL: in ELEM) is
        pl : PLACE;
        begin
        if IDLIST.TVAL'LAST = IDLIST.t +1 then raise SATURATION; end if;
        pl := rang -1;
        if pl >idlist.queue +1 then raise INCOHERENCE; end if ;

        if idlist.t /= 0 then
                -- decalage des elements
                for i in reverse idlist.queue .. pl loop
                        idlist.tval(i+1) := idlist.tval(i);
                end loop;
                -- mise a jour de la queue de la liste
                idlist.queue := idlist.queue+1;
        end if;
        -- adjonction
        idlist.tval(pl) := idval;
        -- modification de la taille de la liste
        idlist.t := idlist.t +1 ;
```

```
        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when SATURATION =>
                put("La liste est pleine; cf declaration et taille");

        end ADJL;


Function TAILLE (IDLIST : in LISTE) return integer is
        begin
        return (idlist.t);
        end TAILLE;


Function EST_VIDE(IDLIST: in LISTE ) return BOOLEAN is
        begin
        return (idlist.t = 0);
        end EST_VIDE;

Function TETE   (IDLIST : in LISTE) return PLACE is
        begin
        return (idlist.tete);
        end TETE;

Function QUEUE  (IDLIST : in LISTE) return PLACE is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;
        return (idlist.queue);

        exception
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");

        end QUEUE;

Procedure SUPL(IDLIST: in out LISTE;RANG: in TRANG) is
        pl : PLACE;
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;

        pl := RANG -1;
        -- decalage des elements
        for i in pl+1 .. idlist.queue loop
                idlist.tval(i-1) := idlist.tval(i);
        end loop;
        -- suppression
        -- modification de la taille de la liste
        idlist.t := idlist.t -1 ;
        -- mise a jour de la queue de la liste
        if (idlist.t /= 0) then
                idlist.queue := idlist.queue-1;
        end if;

        exception
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        end SUPL;

Procedure CHGLP(IDLIST: in out LISTE;PL:in PLACE;IDVAL:in ELEM) is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;
        if PL > IDLIST.QUEUE +1 then raise INCOHERENCE; end if;
        if PL < IDLIST.TVAL'FIRST or PL >IDLIST.TVAL'LAST
                then raise DEBORDEMENT; end if;
```



```
        idlist.tval(pl) := idval;

        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when DEBORDEMENT =>
                put ("Rang ou place hors limites ; cf declaration de liste");
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        end CHGLP;

Function ACCES(IDLIST: in LISTE;RANG:in TRANG) return PLACE is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;
        if RANG > IDLIST.QUEUE +1 then raise INCOHERENCE; end if;
        if RANG < IDLIST.TVAL'FIRST+1 or RANG >IDLIST.TVAL'LAST+1
                then raise DEBORDEMENT; end if;

        return (RANG-1);

        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        when DEBORDEMENT =>
                put ("Rang ou place hors limites ; cf declaration de liste");

        end ACCES;

Function SUCC(IDLIST:in LISTE;PL:in PLACE) return PLACE is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;
        if PL > IDLIST.QUEUE +1 then raise INCOHERENCE; end if;
        if pl = idlist.queue then raise FIN_DE_LISTE;   end if;

        return(pl +1);

        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when FIN_DE_LISTE =>
                put("On atteint la queue de la liste");

        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        end SUCC;

Function PRED(IDLIST :in LISTE;PL:in PLACE) return PLACE is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;
        if PL > IDLIST.QUEUE +1 then raise INCOHERENCE; end if;
        if pl = idlist.tete then raise DEBUT_DE_LISTE; end if;

        return(pl-1);
        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        when DEBUT_DE_LISTE =>
                put("On atteint la tete de la liste");

        end PRED;
```

```
Function VALEUR(IDLIST:in LISTE;PL:in PLACE) return ELEM is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;
        if PL > IDLIST.QUEUE  then raise INCOHERENCE; end if;
        if PL < IDLIST.TVAL'FIRST or PL >IDLIST.TVAL'LAST
                then raise DEBORDEMENT; end if;

        return(idlist.tval(pl));

        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        when DEBORDEMENT =>
                put ("Rang ou place hors limites ; cf declaration de liste");

        end VALEUR;

Procedure PARC_TOTAL(IDLIST: in out LISTE; PL: out PLACE ) is
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;

        for i in idlist.tete .. idlist.queue loop
                pl := i;
                FONC(idlist.tval(i));
        end loop;

        exception
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        end PARC_TOTAL;

Procedure PREM_TEL_QUE(IDLIST:in LISTE;
                       RANG: out TRANG;PL: out PLACE) is
        ind : place;
        arret : boolean := FALSE;
        begin
        if IDLIST.t = 0 then raise LISTE_VIDE; end if;

        for i in idlist.tete .. idlist.queue loop
                ind := i;
                arret := PREDICAT(idlist.tval(i));
                exit when arret;
        end loop;
        if arret then rang := ind+1 ; pl := ind;
                else rang := 0; pl := idlist.queue;
        end if;

        exception
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        end PREM_TEL_QUE;

Procedure AJOUT_APRES(IDLIST: in out LISTE;PL: in PLACE;IDVAL: in ELEM) is
        begin
        if PL > IDLIST.QUEUE +1 then raise INCOHERENCE; end if;
        if PL < IDLIST.TVAL'FIRST or PL >IDLIST.TVAL'LAST
                then raise DEBORDEMENT; end if;
        if idlist.t /= 0 then
                -- decalage des elements
                for i in reverse idlist.queue .. pl+1 loop
                        idlist.tval(i+1) := idlist.tval(i);
                end loop;
```



```
                -- mise a jour de la queue de la liste
                idlist.queue := idlist.queue+1;
        end if;
        -- adjonction
        idlist.tval(pl) := idval;
        -- modification de la taille de la liste
        idlist.t := idlist.t +1 ;

        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        when DEBORDEMENT =>
                put ("Rang ou place hors limites ; cf declaration de liste");

        end AJOUT_APRES;

Procedure COPIE(IDLIST: in LISTE; IDLIST_SORT: out LISTE) is
        begin
        idlist_sort.tete := idlist.tete;
        if idlist.t /= 0 then
                for i in idlist.tete .. idlist.queue loop
                        idlist_sort.tval(i) := idlist.tval(i);
                end loop;
        end if;
        idlist_sort.t := idlist.t;
        idlist_sort.queue := idlist.queue;
        end COPIE;

Procedure CONCAT(IDLIST_DEB:in LISTE;IDLIST_SUIT:in LISTE;
                 IDLIST  : out LISTE) is
        pli,plj : place;
        begin
        if IDLIST_DEB.t = 0 then raise LISTE_VIDE; end if;
        if IDLIST_SUIT.t = 0 then raise LISTE_VIDE; end if;

        idlist.tete := 0;
        for i in idlist_deb.tete .. idlist_deb.queue loop
                idlist.tval(i) := idlist_deb.tval(i);
                pli := i;
        end loop;
        for j in idlist_suit.tete .. idlist_suit.queue loop
                idlist.tval(pli+j+1) := idlist_suit.tval(j);
                plj := j;
        end loop;
        idlist.t := idlist_deb.t + idlist_suit.t;
        idlist.queue := plj + pli +1;

        exception
        when LISTE_VIDE =>
                put("Operation illicite sur liste vide");
        end CONCAT;

Procedure PARC_PARTIEL(IDLIST:in out LISTE;DEB: in PLACE; FIN : out PLACE) is
        ind : PLACE;
        begin
        if DEB > IDLIST.QUEUE +1 then raise INCOHERENCE; end if;

        for i in deb .. idlist.queue loop
                ind := i;
                exit when PREDICAT(idlist.tval(i));
                FONC(idlist.tval(i));
        end loop;
        FIN := ind;
```

```
        exception
        when INCOHERENCE =>
                put("Rang impossible ; cf taille de la liste");
        end PARC_PARTIEL;

end LIS_MANAGER_CTG;
```


Chapitre 2

CONCEPTION DE LA BASE DE CONNAISSANCES DE SAIDA SUR LES CRITERES DE CHOIX DE REPRESENTATION DE DONNEES

Nous venons de présenter la bibliothèque de textes et de paquetages ADA mise à la disposition de l'utilisateur de SAIDA. Nous décrivons maintenant le module d'aide au choix d'un des modules de cette bibliothèque pour un objet abstrait d'algorithme écrit en ADA.

Le système SAIDA doit connaître les critères qui, à partir de l'analyse d'un algorithme et de son environnement, permettent de proposer une ou des représentations efficaces pour un objet de cet algorithme. Dans les ouvrages de programmation, cette expertise est exprimée de façon informelle et non uniforme, souvent à partir d'exemples.

L'objectif de ce chapitre est de montrer comment cette expertise est rassemblée et formulée dans la base de connaissances de SAIDA . Cette base représente a priori le point de vue du programmeur, indépendamment des considérations pédagogiques qui font l'objet du chapitre suivant. Nous avons choisi de décrire certaines étapes de la création de cette base et d'essayer de faire apparaître un fil conducteur dans le travail parce que nous pensons que cela peut aider des utilisateurs non familiers des systèmes experts, notamment des enseignants, à créer leurs propres bases.

Nous avons adopté le mode de travail assez classique suivant : identification des concepts et raisonnements du domaine à partir d'exemples, formalisation, choix d'un mode d'expression et création d'une première version à partir des informations "livresques", validations et modifications sur des exemples. Nous en présentons les aspects généraux dans le premier paragraphe et abordons ensuite successivement le cas des listes et celui des ensembles.

Nous terminons par des suggestions d'extension de cette base de connaissances qui permettraient de prendre en compte une catégorie de problèmes plus vaste que celle qui est envisagée aujourd'hui et par une réflexion sur l'ensemble du processus de construction en montrant comment on en retrouve les principales caractéristiques dans d'autres domaines.

2.1 LES OBJETS ET LES RAISONNEMENTS DE L'UNIVERS

Laurière distingue dans la connaissance nécessaire pour la résolution de problèmes plusieurs niveaux de savoir [LAURI,86b]. Nous en retenons trois :

- Les éléments de base et les abstractions intermédiaires de l'univers, nous les désignerons par **les concepts** de l'univers.
- Les procédés permettant, à partir de certaines informations relatives au problème traité, d'en déduire d'autres ; nous parlerons **de raisonnements élémentaires.**
- Le ou les processus permettant d'enchaîner des raisonnements élémentaires pour résoudre un problème ; nous emploierons le terme de **conduite du raisonnement.**

Nous nous intéressons dans ce paragraphe surtout au premier niveau et en partie au second. En effet, nous voulons d'abord commencer à construire et à structurer l'ensemble de concepts qui nous permettra de décrire chacun des problèmes soumis au système. Un problème pour ce système est toujours de la forme : "Trouver une bonne représentation pour un objet abstrait d'un algorithme ADA". Nous voulons également caractériser les raisonnements élémentaires de ce domaine de travail. Nous nous appuyons sur des exemples pour réaliser cette étude préliminaire. Notre objectif est de disposer au terme de cette étude d'informations suffisantes pour décider d'un mode d'expression de la connaissance dans le système.

2.1.1 Représentation des objets de la procédure COMPTE-MOTS

Nous reprenons ici l'exemple de la procédure COMPTE-MOTS de la première partie (chapitre 1).

Objet TEXTE

TEXTE est déclaré dans cette procédure comme LISTE (CHAINE) où CHAINE est un type représentant des chaînes de caractères. Il s'agit donc de représenter un objet de type LISTE.

Hypothèse 1 : L'objet TEXTE est très volumineux et on sait qu'on dispose de trop peu de place en mémoire centrale, on va donc le représenter sur support externe, par exemple sur disque. La seule opération effectuée sur cet objet est un parcours séquentiel. Avec ces hypothèses, TEXTE peut être représenté par un fichier séquentiel.

Pour parvenir à la conclusion on a utilisé des concepts de l'univers de travail :

- la taille de l'objet avec le qualificatif "très volumineuse"
- la place disponible en mémoire centrale avec le qualificatif "trop peu"
- le support de l'information avec le qualificatif "externe"

et un raisonnement élémentaire qui peut s'exprimer de la façon suivante :

si la taille de l'objet est très grande
et il n'y a pas assez de place en mémoire centrale
alors le support sera externe.

Dans un deuxième temps de la résolution, est apparue la caractéristique "opération" avec :

- opération PARCOURS
- pas d'autres opérations

et un raisonnement qui peut être formulé ainsi :

si le support est externe,
et si la seule opération effectuée sur la liste est un parcours
alors l'implantation conseillée est un fichier séquentiel.

Hypothèse 2 : L'objet TEXTE peut être implanté en mémoire centrale. Si on connait la taille maximum de cet objet, on peut le représenter dans un tableau.
Si au contraire la taille de cet objet varie beaucoup selon les exécutions et si TEXTE coexiste en mémoire avec d'autres objets de taille variable, une représentation chaînée par pointeurs serait meilleure et sans incidence sur le temps de calcul puisqu'on ne fait sur TEXTE qu'un parcours séquentiel.

De nouveaux concepts ont été utilisés :

- taille maximum de l'objet qualifiée de fixe ou très variable,
- coexistence de plusieurs objets de taille variable en mémoire,
- temps de calcul.

Le concept implantation a reçu successivement comme valeurs :

- fichier séquentiel,
- tableau,
- chaînage par pointeurs.

Nous allons maintenant étudier la représentation de l'autre objet abstrait de cette même procédure, MOTS-CLES, déclaré comme ENSEMBLE et enrichir notre univers des concepts.

objet MOTS-CLES

MOTS-CLES est déclaré dans la procédure COMPTE-MOTS comme ENSEMBLE de CHAINE.

Ce qui caractérise à l'évidence l'objet MOTS-CLES, c'est qu'on en consulte souvent le contenu, plus précisément on appelle la fonction APPARTIENT (MOTS-CLES, E) à l'examen de chaque mot du texte. On va donc choisir une représentation qui optimise le temps d'accès à un élément de l'ensemble et la place mémoire nécessaire pour représenter cet objet si c'est possible. Sinon on peut soit proposer à l'utilisateur de nous donner une information complémentaire - veut-il privilégier le temps d'exécution de son programme par rapport à la place mémoire utilisée ? - soit proposer une solution qui réalise un bon compromis, mais l'existence d'une bonne fonction de hachage et une représentation en adressage dispersé permettrait des accès plus rapides aux éléments.

Nouveaux concepts utilisés (venant compléter la liste précédente) :

- fonction APPARTIENT utilisée souvent
- privilégier le temps de calcul sur la place mémoire nécessaire
- existence d'une bonne fonction de hachage.

2.1.2 Analyse des concepts et raisonnements rencontrés

Les concepts

Un premier examen des concepts rencontrés dans les exemples précédents amène à deux remarques :

- Les concepts mentionnés ne sont pas tous de même nature : certains sont liés à des contraintes extérieures à l'algorithme lui-même, comme un support physique imposé, d'autres concernent l'objet lui même, sa taille par exemple, d'autres encore font appel aux opérations utilisées dans l'algorithme sur l'objet, etc... Il n'est pas souhaitable de continuer à accumuler des informations de natures aussi diverses et il y a intérêt, pour poursuivre le processus de recueil de la connaissance, à chercher un mode de structuration qui permette de l'organiser.

- Ces concepts n'ont, pour l'instant, pas d'autre sémantique que celle que leur libellé en français peut leur conférer ; leur définition doit être affinée notamment pour préciser dans quels ensembles certains d'entre eux prennent des valeurs.

Structuration de l'univers des concepts

La connaissance que nous avons du domaine de travail, notamment à partir de l'étude des manuels, nous amène à proposer la structuration suivante pour l'univers des concepts :

. Contraintes extérieures à l'algorithme
- support de l'information
- rémanence de l'information
- temps d'exécution
- espace mémoire disponible
- ...

. Caractéristiques des objets de l'algorithme
(celui que l'on veut représenter, désigné par "l'objet", et les autres)
- existence d'autres objets
- taille maximum de l'objet
- variations de la taille de l'objet
- ordre sur les éléments
- taille d'un objet
- ...

. Caractéristiques des traitements effectués sur l'objet à représenter
- utilisation d'un groupe d'opérateurs

œ, U, D, - , C (opérations ensemblistes)

- fréquence d'un groupe d'opérations par rapport à un autre
- ...

D'autres découpages de l'univers des concepts sont évidemment possibles. Il s'agit pour l'instant seulement d'organiser le recueil des informations, de pouvoir, lorsqu'une notion est rencontrée, savoir plus facilement s'il s'agit d'un concept nouveau ou d'une nouvelle valeur pour un concept déjà répertorié. Cependant ces "titres de chapitres" jouent aussi un rôle au niveau de l'expression des raisonnements élémentaires, nous en avons rencontré un exemple avec le concept "support" que nous avons utilisé dans le problème "COMPTE-MOTS, et au niveau de la conduite de la résolution des problèmes. Nous reviendrons donc sur leur rôle dans la suite de ce chapitre et dans le suivant.

D'un concept à un ensemble de valeurs associées

Prenons l'exemple du concept "ajouter un élément" à la structure de données que l'on étudie, notamment dans le cas d'une liste ; à propos de ce concept, nous relevons dans nos descriptions informelles de la connaissance les expressions suivantes :

"il existe des adjonctions"
"il existe beaucoup d'adjonctions"
"il n'existe que des adjonctions"
"il existe peu d'adjonctions"
"les adjonctions ne sont effectuées qu'en tête"
"il existe p adjonctions" (à comparer avec q suppressions par exemple).

Nous avons besoin d'exprimer ces différentes nuances, et nous notons donc que pour ce concept d'adjonction, nous avons selon les contextes plusieurs ensembles de valeurs possibles.

Dans certains cas, il s'agit de quantifier cette notion d'adjonction, et l'échelle des valeurs utilisées peut être :

- rudimentaire [VRAI, FAUX],
- plus élaborée [PEU, MOYEN, BEAUCOUP],
- un intervalle d'entiers [1 .. N].

Dans d'autres cas, il s'agit de noter l'existence d'une propriété :

- adjonctions en tête de liste seulement
- adjonctions à l'exclusion d'autres opérations.

Retenons simplement que beaucoup de caractéristiques différentes peuvent être attachées à un même concept selon le contexte dans lequel il est utilisé. Il nous faudra soit choisir un mode d'expression des concepts qui permette cette diversité, soit décomposer ce concept en éléments plus fins.

Une démarche de modélisation

Ces choix de concepts et de structuration d'univers rappellent ceux dont il est question pour la création du schéma conceptuel d'un système d'informations [BEN,79], [FOU,82],ou pour la description d'un domaine d'enseignement en termes d'items ou de concepts et de relations entre concepts [QUE,80]. Il s'agit de la démarche générale de

modélisation d'un univers.

Dans les bases de données, le choix des concepts influe sur la plus ou moins grande facilité avec laquelle les requêtes peuvent être exprimées. Dans les bases de connaissances, le choix des concepts conduit à des règles plus ou moins faciles à exprimer, plus ou moins lisibles pour l'utilisateur.

Une dernière remarque avant de parler des raisonnements

La modélisation d'univers ainsi effectuée peut supporter des raisonnements différents pour résoudre des problèmes de nature différente. Dans l'environnement des algorithmes, des raisonnements sur la complexité et les performances auraient probablement pour base une modélisation d'univers voisine de celle que nous effectuons. C'est sans doute une raison supplémentaire pour essayer de bien séparer univers des concepts et raisonnements opérés.

Les raisonnements

Les exemples que nous venons de présenter n'ont pas permis une analyse fine des raisonnements mis en oeuvre. Cependant, nous pouvons observer que nous avons facilement exprimé un "raisonnement élémentaire" sous la forme :

Si liste de conditions alors conclusion

et la résolution des problèmes par une suite de raisonnements élémentaires de ce type.

La conclusion finale consiste à proposer une implantation en mémoire pour l'objet étudié ; dans certains cas plusieurs représentations peuvent être suggérées, mais il n'y a pas de facteurs d'incertitude sur l'une ou l'autre d'entre elles comme par exemple dans un diagnostic médical de MYCIN où l'on peut conclure à la maladie M avec 60% de chances de succès. Il ne semble donc pas a priori que nous soyons dans un univers de raisonnement incertain.

Le domaine de travail peut être caractérisé par une réalité complexe nécessitant beaucoup de concepts pour rendre compte de raisonnements peu profonds, par une forte imbrication de niveaux et de critères et par des connaissances éparses et disparates à rassembler.

Cette rapide analyse des concepts et raisonnements utilisés pour résoudre des problèmes de représentation d'objets abstraits nous permet de passer maintenant au choix

d'un mode d'expression de la connaissance.

2.1.3 Choix d'un mode d'expression de la connaissance

Les choix possibles

Laurière, dans [LAURI,86b], propose une liste des procédés utilisés aujourd'hui dans les systèmes experts pour représenter la connaissance. On y trouve, en allant du plus procédural au plus déclaratif :

automate fini
programme
script
réseau sémantique
prototype (frame)
graphe, réseau
spécification formelle
calcul des prédicats
théorèmes et règles de réécriture
règles de production
phrases du langage naturel.

La solution idéale consiste donc sans doute à analyser l'adéquation de chacune de ces propositions au domaine dans lequel on veut travailler. C'est tout à fait irréaliste pour au moins deux raisons :

On s'exprime mieux dans un formalisme que l'on a étudié à fond et dont on a une bonne pratique que dans un autre ; pour faire un choix relativement neutre, il faudrait donc avoir une excellente connaissance et une pratique suffisante de tous les procédés ci-dessus, condition qui nous semble rarement remplie !

Nous ne pouvons, même au stade de la réflexion conceptuelle, oublier que nous allons utiliser des logiciels et des machines, et on dispose rarement de l'accès à beaucoup d'environnements opérationnels de construction de systèmes experts.

Nous avons donc adopté une attitude pragmatique courante aujourd'hui : vérifier que l'environnement de développement que l'on connait et dont on dispose effectivement permet de décrire les connaissances de l'univers que l'on vient d'analyser.

La solution retenue

C'est la raison qui nous a conduit à développer un système à règles de productions, mais d'autres cadres auraient pu être retenus, notamment une réalisation dans un environnement permettant une description de l'univers en termes d'objets.

Dans un système à règles de productions, les raisonnements élémentaires sont représentés par une règle de production, c'est-à-dire une expression de la forme :

MG ----> MD

dans laquelle le membre gauche MG décrit une certaine situation, représentée dans un formalisme adapté à l'univers de travail, et où le membre droit MD donne l'action à envisager lorsque la situation décrite en MG est rencontrée [LAURI,86b]. Le mode de raisonnement élémentaire sous-jacent est évidemment le modus ponens.

Exemple de règle :

si le type de l'élément est entier ou booléen ou réel ou caractère alors les éléments sont de type simple et la taille d'un élément est petite.

Cette description s'applique à tout système de production. Il nous reste à préciser comment nous décrivons les concepts de l'univers et quelles formes nous donnons aux membres gauche et droit des règles. Nous nous contentons ici d'une description informelle illustrée par des exemples, la grammaire du langage utilisé est donnée dans la quatrième partie de la thèse.

Le langage utilisé est une extension de la logique des propositions. En logique des propositions, un élément de connaissance est représenté par une "proposition", par exemple il-existe-des-adjonctions, qui peut être vraie ou fausse. Nos propositions élémentaires sont des égalités d'un attribut à une valeur ou à un autre attribut. Un concept est donc représenté par un attribut et un ensemble de valeurs possibles pour cet attribut ; l'univers d'un problème est modélisé par un ensemble de couples (attribut, valeur).

Voici, à titre d'exemple l'état de l'univers du problème des MOTS-CLES lors de l'implantation de l'objet TEXTE ; dans chaque couple la première chaîne de caractères est un nom d'attribut, la seconde l'une des valeurs possibles pour cet attribut :

(taille-de-l'objet, très-volumineuse)
(place-mémoire-centrale, trop-peu)
(support, externe)
(opération-parcours, vrai)

(pas-d'autre-opération-que-parcours, vrai)
(implantation, fichier-séquentiel)

Nous disposons maintenant d'un cadre pour construire la base de connaissances de SAIDA. Nous allons successivement décrire la partie de la base relative à l'implantation d'objets de type LISTE, puis celle relative aux objets de type ENSEMBLE. Nous avons choisi des approches différentes ; le système propose une seule solution dans le cas des listes, plusieurs dans le cas des ensembles. Ce choix pourrait a priori sembler nuire à l'homogénéité de la base ; il nous a semblé au contraire intéressant de montrer deux réalisations différentes qui peuvent se justifier par une plus grande variété de représentations pour les ensembles que pour les listes.

2.2 LA BASE DE CONNAISSANCES RELATIVE AUX LISTES

Dans l'étude précédente, nous nous sommes contentés d'informations issues de la résolution de quelques problèmes d'une part et de manuels d'autre part. Nous décrivons maintenant une base de connaissances permettant de résoudre de nombreux problèmes d'implantation d'objets de type LISTE.

Un premier sous-paragraphe précise le type de problème soumis au système. Nous présentons ensuite les connaissances utilisées selon les niveaux que nous avons définis : concepts, raisonnements élémentaires, conduite de la résolution d'un problème. Dans la réalité ces niveaux sont évidemment explorés en parallèle.

La question du choix d'implantation d'un objet de type LISTE n'offrant pas beaucoup de difficultés, nous en profitons pour détailler certains aspects de la construction des règles comme l'introduction de "concepts intermédiaires".

2.2.1 Les problèmes soumis au système, les implantations retenues.

Comme nous l'avons indiqué en 2.1, nous appelons problème soumis au système le choix d'une implantation, parmi celles contenues dans la bibliothèque de SAIDA, pour un objet abstrait de l'algorithme que le programmeur est en train d'écrire à l'aide du système SAIDA.

De plus, nous faisons l'hypothèse de réaliser l'implantation d'un seul objet abstrait à la fois, sans liens explicites avec les solutions proposées pour d'autres objets. Toutefois certaines règles prennent en compte l'existence d'autres objets et demandent à l'utilisateur des

informations à leur sujet.

C'est une limitation actuelle du système, mais c'est une phase nécessaire ; nous mentionnons dans les prolongements souhaitables du système la possibilité d'un niveau de raisonnement prenant ensuite en compte l'ensemble des objets à représenter et des solutions proposées.

Pour les listes, les implantations actuellement prises en compte dans la base de connaissances sont les suivantes :

- sur support externe
  - fichier à accès direct
  - fichier séquentiel

- sur support interne
  - en représentation contiguë
    - tableau non trié des éléments
    - tableau trié des éléments
  - en représentation chaînée avec pointeurs
    - simple chaînage non trié
    - simple chaînage trié
    - double chaînage non trié
    - double chaînage trié

La structuration proposée pour ces implantations met en évidence les notions de support de l'information, de contiguïté et chaînage en mémoire et d'organisation des informations.

2.2.2 Les concepts de l'univers

Quand on construit un système à règles de production, on écrit des règles, on en modifie, on en ajoute, on en supprime. Les règles les plus simples comportent en partie gauche des caractéristiques du problème (algorithme, objet lui-même, contraintes externes) et en partie droite la proposition d'une représentation.

Pour représenter ces règles, il suffit de recenser toutes les caractéristiques du problème qui peuvent être utiles, de leur associer concepts et valeurs permettant de les exprimer et d'enrichir ainsi l'ensemble des concepts initialisé lors de l'étude préliminaire. Un concept particulier, nommé implantation et prenant pour valeurs toutes les représentations

citées en 2.2.1, permet d'écrire les parties droites des règles. La figure suivante donne la liste des concepts liés à l'énoncé du problème, que nous distinguons des concepts liés à la résolution du problème comme implantation.

Concepts relatifs aux contraintes extérieures

nom	valeurs
support	interne, externe
	la liste existe dans un fichier
	vrai faux
	obligation d'utiliser un fichier
	vrai faux
	rémanence de la liste
	vrai faux
	temps d'exécution à minimiser
	vrai faux
	existence de problèmes de place en mémoire
	vrai faux

Concepts liés aux objets de l'algorithme abstrait

* liés à la taille de l'objet à implanter

nom	valeurs
la taille de la liste est bornée	vrai faux
nombre d'éléments de la liste	petit, moyen, grand.
taille de la liste	grande, moyenne, petite
variation du nombre d'éléments	varie peu, varie, varie beaucoup

*liés à l'état de la liste vis à vis de l'algorithme

la liste possède une valeur initiale	vrai faux

*liés à l'organisation des éléments dans la liste

il existe une relation d'ordre sur les éléments	vrai faux
on s'interdit de modifier l'ordre des éléments	vrai faux
la relation d'ordre sur les éléments est utilisée dans les parcours	vrai faux

*liés aux éléments de la liste

taille d'un élément	fixe variable

Concepts liés aux traitements effectués dans l'algorithme

* liés à l'énoncé d'un nom d'opération

* adjonctions

- il y a des adjonctions	vrai	faux
- il n'y a que des adjonctions	vrai	faux
- il y a beaucoup d'adjonctions	vrai	faux
- les adjonctions ont lieu exclusivement en tête	vrai	faux

* suppressions (mêmes concepts que pour les adjonctions)

* modifications (mêmes concepts que pour les adjonctions)

* accès

* liées à des notions plus larges que celle d'opérateur (rang, adj/supp,...)

- beaucoup plus d'accès que d'adjonctions / suppressions
- beaucoup de recherches associatives
- parcours à partir de la queue

Mais toutes les règles ne sont pas de cette forme. Lorsque la résolution du problème demande une analyse fine de nombreux paramètres, la règle correspondante aurait un trop grand nombre de prémisses ou des prémisses trop longues : dans ce cas, on définit un intermédiaire.

Supposons qu'on veuille exprimer le fait que les éléments de la liste sont de l'un des types suivants : entier, booléen, réel , caractère, énuméré.

Exemple : Si type-élément = entier ou booléen ou réel ou caractère ou énuméré
alors type-simple

L'attribut type-simple est vrai si cette propriété est vérifiée, on l'utilise dans d'autres règles comme prémisse. Il appartient à l'univers du problème et peut figurer comme tête de sous-rubrique dans notre classification précédente.

Supposons maintenant qu'on ait déterminé les caractéristiques suivantes pour un objet de type LISTE :

support = interne
il existe des adjonctions/suppressions
il existe des recherches associatives
il existe une relation d'ordre sur les éléments

On peut proposer une représentation chaînée triée ou une représentation contiguë triée. Pour trancher, d'autres informations doivent intervenir, par exemple :
L'utilisateur veut-il privilégier les adjonctions/suppressions ?
Y a-t-il des problèmes de place en mémoire ?
La taille de la liste est-elle bornée ?

Si on n'utilise pas d'intermédiaires, on va écrire beaucoup de règles ayant chacune de nombreuses prémisses.

2.2.3 Les raisonnements élémentaires

On ne peut indéfiniment multiplier le nombre des règles, ni accroître le nombre de leurs prémisses sans nuire à la lisibilité de l'ensemble ; on a besoin d'exprimer d'autres règles avec des concepts intermédiaires. Ces concepts nécessaires à l'expression de raisonnements font aussi partie de l'univers de travail, mais ils sont liés au processus de résolution du problème et nous les distinguons de ceux liés à l'énoncé, indépendamment de tout raisonnement.

* Intermédiaires traduisant une étape dans le raisonnement

Exemple : si les variations du nombre d'éléments = varie beaucoup
alors représentation 1 = chaînée.

Le concept *représentation 1* traduit une première orientation de représentation, ici vers un chaînage, orientation qui demande d'une part à être confirmée, d'autre part à être affinée. En effet, nous scindons là le concept implantation en deux concepts particuliers, la représentation chaînée ou contiguë et l'organisation triée ou non triée ; cette décomposition est évidemment inspirée par une structuration possible de l'ensemble des représentations que nous avons mise en évidence précédemment.

Pour confirmer : Une autre idée de première orientation est évidemment une représentation contiguë, on pourrait donc penser à un concept intermédiaire ayant pour valeurs possibles contiguë et chaînée.

C'est une solution qui conduirait à des contradictions, nous venons de voir que certaines caractéristiques peuvent plaider pour la contiguïté, d'autres pour le chaînage. La

solution retenue ici est celle de deux concepts :
représentation 1 prend la valeur chaînée si des éléments conduisent à une représentation de cette catégorie, inconnue sinon.
représentation 2 prend la valeur contiguë si des éléments conduisent à une représentation de cette catégorie, inconnue sinon.

S'il n'y a pas de conflit, l'orientation est confirmée par l'une des règles :
Exemple : si représentation 1 = chaînée, représentation 2 = inconnue
alors représentation = chaînée

si représentation 2 = contiguë, représentation 1 = inconnue
alors représentation = contiguë

S'il y a conflit, la valeur de représentation sera déduite d'autres règles prenant en compte d'autres caractéristiques :

Exemple : si représentation 1 = chaînée,
représentation 2 = contiguë,
beaucoup plus d'accès que d'adjonction/suppression,
on souhaite privilégier les accès
alors représentation = contiguë

Le concept *représentation* qui prend pour valeur contiguë ou chaînée permet de confirmer les orientations des concepts représentation 1 et représentation 2.

Pour affiner
Il nous faut maintenant des indications permettant d'affiner le résultat obtenu. Le concept *organisation* à valeurs dans (triée, quelconque) va permettre de déduire, indépendamment des aspects contigu ou chaîné, l'organisation souhaitée pour l'objet .

Exemple : si il existe une relation d'ordre sur les éléments,
il y a beaucoup d'accès,
on s'autorise à modifier l'ordre des éléments
alors organisation = triée

* Intermédiaires nécessaires à la résolution de certains problèmes

Nous voulons exprimer l'idée que dans certains cas, même si la représentation de l'objet étudié est à priori imposée sur support externe parce que fournie ainsi, il peut être

judicieux de réaliser les traitements décrits par l'algorithme en transférant les données en mémoire centrale. Cette idée est traduite par la règle :

Exemple : si on doit minimiser le temps d'exécution,
la place mémoire est suffisante,
la liste existe dans un fichier,
alors adjonction début algorithme = transfert fichier mémoire centrale,
adjonction fin algorithme = transfert fichier externe,
support = interne

Nous venons de créer les deux intermédiaires :
adjonction début algoritme
adjonction fin algorithme
qui, au moment de l'affichage de la valeur de implantation, serviront à donner à l'utilisateur des indications pour modifier l'algorithme s'il retient cette implantation.

2.2.4 La conduite des raisonnements

La résolution d'un problème nécessite un dialogue avec l'utilisateur pour l'acquisition par le système des caractéristiques du problème utiles à l'élaboration de la solution. Le confort de l'utilisateur et la vision plus ou moins "intelligente" que le système donne de son comportement dépendent donc beaucoup de ce dialogue.

Un bon dialogue doit remplir au minimum les trois conditions suivantes :

- Ne pas contenir de question apparemment "stupide".
Exemple : si l'utilisateur a répondu qu'il y avait beaucoup d'adjonctions dans l'objet de type LISTE étudié, il ne faut pas lui demander ensuite s'il existe des adjonctions ; cette dernière information doit être obtenue par application d'une règle de type suivant :
si il existe beaucoup d'adjonctions
alors il existe des adjonctions
Nous avons ainsi un exemple d'une nouvelle catégorie de règles qu'il faut ajouter au système.

- Poser le moins de questions inutiles possible.

- Amener les questions dans un ordre obéissant à une certaine logique.

Ces deux derniers points relèvent d'une stratégie de raisonnement qu'il faut définir.

Stratégies de raisonnement

On ne raisonne pas en combinant des déductions au hasard, on est en général guidé par un fil conducteur, une "stratégie". Parmi des stratégies possibles, citons :

- Un raisonnement dirigé par les données.
Cela conduit dans cette application à un recueil plus ou moins systématique de caractéristiques du problème dans un ordre à "optimiser" pour le confort de l'utilisateur. Mais, nous avons vu dans les quelques exemples analysés que très souvent on n'a pas besoin pour conclure de toutes les caractéristiques prévues dans le modèle.

- Un raisonnement dirigé par les résultats.
Cela nous conduit ici à envisager chaque représentation possible et à rechercher si les caractéristiques qui peuvent conduire à cette représentation sont présentes.

- Un raisonnement privilégiant certaines représentations.
On peut privilégier certaines représentations pour lesquelles on arrive à conclure assez vite, puis adopter l'une ou l'autre des stratégies précédentes.

- etc...

Expression des stratégies de raisonnement

Dans les systèmes à règles de productions, le raisonnement est conduit par le "moteur d'inférences" ainsi nommé parce qu'il permet l'enchaînement de l'application des règles. Certains moteurs ont une stratégie unique de raisonnement, connue ou inconnue de l'utilisateur, et ne laissent aucune possibilité d'intervention à ce niveau.

Pour un problème de représentation d'objet, un système fonctionnant en chaînage arrière pur reçoit le nom de l'attribut pour lequel on recherche une valeur, ici "implantation", va utiliser toutes les règles qui ont implantation en conclusion, vérifier si leurs prémisses sont satisfaites et sinon se fixer comme but intermédiaire de les satisfaire... On arrive ainsi à des prémisses qui sont "demandables" et aux questions posées à l'utilisateur. Le dialogue est lié à l'ordre d'application des règles par le moteur.

D'autres moteurs vont chercher une part de leur stratégie de raisonnement dans des règles spécialement écrites à cet effet. L'idéal serait que le moteur soit un simple interprète de descriptions de raisonnements. Cela permet au concepteur du système d'exprimer non seulement des règles qui lient des valeurs d'attributs entre elles mais aussi d'autres règles qui

indiquent dans quel ordre et à quelles conditions on veut appliquer les précédentes. Ces derniers sont parfois nommés "métarègles" [LAURI,86b] ou métaconnaissances.

Nous avons utilisé pour guider le raisonnement **la notion d'étape**. Dans une étape, nous cherchons à valuer systématiquement les attributs attachés à certains concepts intermédiaires et à travailler en distinguant par exemple un choix entre représentation en mémoire centrale ou non, puis représentation contiguë ou chaînée, puis selon les cas organisation triée ou non. Chaque étape peut être organisée en sous-étapes ; au niveau d'une étape, c'est le moteur du système qui reprend le contrôle de l'ordre d'application des règles et donc du dialogue à conduire avec l'utilisateur.

Exemple de mise en oeuvre

La stratégie que nous venons de décrire conduit au dialogue suivant pour le choix d'une représentation de la liste des données dans le problème du hit-parade énoncé au chapitre 1 de la première partie et pour la solution duquel un programme ADA est fourni en annexe. On autorise dans ce dialogue les réponses "je ne sais pas".

Etape support

Le but de cette étape est de choisir entre support interne ou externe

Saida : Quelle est la valeur de - la place en mémoire - ?
Utilisateur : Je ne sais pas
S : Avons-nous - des problèmes de place en mémoire centrale - ?
U : Non
S : Avons nous - obligation d'utiliser un fichier - ?
U : Non

L'étape est terminée et a permis de conclure support = interne

Etape chaînage

Le but de cette étape est de savoir si on va utiliser une représentation chaînée ou non.

S : Avons-nous - des opérations d'adjonction - ?
U : Non
S : Avons-nous - des opérations de suppression - ?
U : Non

L'étape est terminée et a permis de conclure représentation = contiguë

Etape tri

Le but de cette étape est de savoir si les éléments de la liste doivent être triés
S : Avons-nous - On s'autorise à modifier l'ordre des éléments - ?
U : Je ne sais pas
S : Avons-nous - Il y a beaucoup de recherches associatives - ?
U : Non
S : Avons-nous - On peut établir une relation d'ordre sur les éléments - ?
U : Non

L'étape est terminée et a permis de conclure organisation = quelconque.

Le système fournit alors comme résultat :
implantation = tableau contigu quelconque.

La liste complète des règles qui ont servi à la conduite de ce dialogue ainsi que la trace de la session effective sont donnés en annexe, leur lecture nécessite en effet la compréhension des notations propres au langage et au moteur de MORSE, le générateur de système à base de connaissances utilisé, notations décrites en partie IV.

Nous nous sommes simplement attachés à obtenir du système des réponses satisfaisantes et à créer un dialogue structuré acceptable par l'utilisateur. D'autres formes de dialogue et de raisonnement sont évidemment possibles ; nous en étudions quelques-unes au chapitre suivant sous l'angle de leur intérêt pédagogique. L'ensemble de règles ainsi constitué reste perfectible et facilement modifiable.

2.3 LA BASE DE CONNAISSANCES RELATIVE AUX ENSEMBLES

Par rapport au travail fait pour les listes, la construction de la base relative aux ensembles présente beaucoup de similitudes. L'univers des problèmes est le même ; certains concepts peuvent être réutilisés, la structuration des concepts reste semblable. Les raisonnements élémentaires et la conduite des raisonnements sont de même type. Cependant, parmi les représentations que nous proposons pour des objets de type ensemble, on en trouve souvent plusieurs acceptables. Nous allons donc demander au système de nous donner plusieurs choix.

Comme dans le cas des listes, nous précisons les problèmes que le système aide à résoudre, puis nous décrivons quelques aspects de la construction de cette base.

2.3.1 Les solutions attendues du système

Pour un objet de type ENSEMBLE, la bibliothèque de types contient actuellement neuf implantations. Nous venons d'indiquer que, souvent, plusieurs implantations sont possibles ; nous souhaitons que le système les fournisse, avec une indication de pertinence pour chacune d'elles, car l'utilisateur peut avoir besoin de plusieurs propositions. Prenons, par exemple, le cas des opérations à plusieurs arguments de type ENSEMBLE ; elles ne sont programmées que dans le cas où tous les ensembles ont la même implantation. Il faut alors choisir une implantation acceptable pour les différents ensembles arguments d'une opération. Le nombre de représentations effectivement proposées peut varier et dépend des stratégies de raisonnement implantées dans le système.

Comme pour les listes, le système suggère des implantations pour un objet à la fois ; certaines règles indiquent à l'utilisateur qu'il doit tenir compte de la représentation d'autres objets éventuels.

2.3.2 L'univers des implantations : contenu et degré de pertinence d'une implantation

Le contenu

Les neuf implantations disponibles sont celles décrites dans la bibliothèque de types, c'est à dire:

- un tableau de booléens indexé par le type des éléments de l'ensemble,
- un tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément
- un tableau des valeurs non trié,
- un tableau des valeurs trié,
- un fichier à accès direct,
- des sous-listes chaînées avec une détermination de sous-liste par fonction de hachage,
- un arbre binaire de recherche,
- une liste chaînée de valeurs en ordre quelconque,
- une liste chaînée triée des valeurs.

Le degré de pertinence de chaque implantation

Nous venons de souligner l'intérêt de proposer plusieurs implantations pour un même objet. Il n'est alors pas possible de transposer aux ensembles la modélisation de l'univers des implantations adoptée pour les listes, un attribut "implantation" pouvant prendre

autant de valeurs différentes que d'implantations connues du système. Nous proposons de représenter cette partie d'univers relative aux résultats à obtenir par neuf attributs (autant que d'implantations différentes), chacun d'eux prenant pour valeur l'un des éléments d'une liste de niveaux de pertinence. Le jeu complet de valeurs comprend souhaité, possible, choisi par défaut, refusé.

Souhaité est obtenu lorsqu'un enchaînement de déductions conduit sans hésitation à la représentation considérée.

Exemple : si vers-chaînée,
on souhaite favoriser les opérations accès recherche,
il existe une bonne fonction de hachage
alors liste chaînée avec hachage = souhaitée

Possible est utilisé lorsque la conduite du raisonnement a fait apparaître des informations incomplètes ou des conflits.

Exemple : si il existe une borne connue de la taille de l'ensemble
les variations du nombre d'éléments = varie ou varie beaucoup
éléments homogènes
alors tableau de valeurs des éléments = possible

La valeur possible pour le concept intermédiaire tableau de valeurs des éléments peut provenir de deux types de règles. Dans la première, les variations de taille de l'ensemble en cours d'exécution ne sont pas connues (réponse "je ne sais pas"), dans la seconde, ces variations sont connues comme importantes, ce qui n'exclut pas un tableau de valeurs mais ne constitue pas une orientations prioritaire en ce sens.

Règle 1 : si il existe une borne connue de la taille de l'ensemble,
la taille de l'ensemble varie en cours d'exécution,
les éléments sont homogènes
alors tableau de valeurs des éléments = possible

Règle 2 : si il existe une borne connue de la taille de l'ensemble,
la taille de l'ensemble varie en cours d'exécution = inconnue
les éléments sont homogènes
alors tableau de valeurs des éléments = possible

Choisie par défaut est la valeur retenue lorsque le système ne dispose d'aucune des valeurs souhaité ou possible pour les représentations. Cette situation se produit lorsque l'utilisateur ne donne pas assez d'informations au système, elle conduit toujours à une représentation

chaînée qui peut être triée ou non, selon qu'on sait valuer le concept organisation ou non. Elle est prévue pour qu'on ait toujours une solution.

Exemple : si organisation = inconnue
alors organisation = ordre quelconque

si organisation = ordre quelconque,
chaînage des valeurs = inconnu,
aucune conclusion souhaitée ou possible
alors chaînage des valeurs ordre quelconque = choisie par défaut

Refusé est employé soit lorsque le choix d'une représentation est techniquement impossible, soit lorsque nous l'avons considéré comme déraisonnable. L'utilisateur peut toujours interroger le système, prendre connaissance des règles appliquées et savoir quelle est l'origine du refus.

Exemple de refus par impossibilité technique :
si vers-chaîné,
on veut minimiser la place mémoire occupée
alors arbre binaire de recherche = refusé

Exemple de refus comme représentation "déraisonnable" :
si un support externe n'est pas imposé,
la taille de l'ensemble est compatible avec la place disponible en mémoire (au moins pour certaines représentations)
alors fichier à accès direct = refusé

Le nombre de valeurs proposées pour une représentation

Une représentation comme le tableau de booléens est toujours à retenir si elle est possible puisqu'elle optimise à la fois le temps d'exécution de bon nombre d'opérations, avec une exception pour les parcours, et la place mémoire utilisée. Nous faisons ici l'hypothèse que ces tableaux sont effectivement implantés comme chaînes de bits et ne traitons pas du problème de la taille maximum généralement imposée par les systèmes à ce type de représentations. Pour les tableaux de booléens, le système ne propose donc que souhaité ou refusé.

Dans les autres cas, nous retenons souhaité, possible, refusé avec une exception pour les représentations en liste chaînée de valeurs où la distinction entre souhaité et possible

n'a pas semblé pertinente, et qui n'est pas refusé puisque c'est le choix fait par défaut.

Dans les règles que nous venons de donner comme exemples, nous avons utilisé des intermédiaires de raisonnements et des valeurs pour ces intermédiares dont nous expliquons maintenant la signification.

2.3.3 L'univers des implantations : structuration

Les règles qui ont constitué le premier noyau d'expérimentation ont été construites en analysant les représentations proposées, en les regroupant par familles et en recensant des caractères du problème qui pouvaient orienter vers cette famille de représentations.

A partir des neuf implantations retenues, nous avons structuré l'univers des implantations en familles et défini des attributs permettant d'exprimer des raisonnements s'appuyant sur cette structuration de la façon suivante :

Famille:	Attribut :
utilisation d'un tableau de booléens	vers-booléens
utilisation du tableau des valeurs	vers-tableau de valeurs
utilisation de chaînage par pointeurs	vers-chaîné
utilisation d'un fichier à accès direct	fichier accès direct

Pour chaque famille, nous avons rassemblé les caractéristiques des problèmes qui pouvaient amener à s'orienter vers une représentation de cette famille. Les "raisonnements élémentaires" ainsi recensés sont représentés en utilisant comme noms d'attributs les quatre noms ci-dessus et comme valeurs possibles : souhaité, possible, refusé ou bien vrai, faux.

Exemples de telles règles :
si les éléments du référentiel de l'ensemble sont connus,
la taille de l'ensemble est voisine de celle du référentiel
alors vers-booléens

Les caractéristiques du problème peuvent évidemment orienter le système vers plusieurs familles ; cette situation ne crée pas ici un conflit comme dans le cas des listes, puisque nous acceptons plusieurs représentations. Ces orientations sont ensuite affinées :

. soit pour conclure directement avec, par exemple, la règle suivante :
si vers-chaînée,
on souhaite favoriser les opérations de recherche,
il n'existe pas de bonne fonction de hachage,
il n'y a pas de problèmes de place en mémoire centrale

alors arbre binaire de recherche = souhaité

. soit en utilisant d'autres intermédiaires (tableau de booléens, tableau de valeurs des éléments) avec par exemple la règle ci-après :

si vers-booléens,
élément peut servir d'index dans un tableau en ADA,
la taille de l'ensemble est voisine de celle du référentiel
alors tableau de booléens = souhaité

Il nous reste à expliquer maintenant comment nous proposons d'enchaîner l'application de ces règles dans la résolution d'un problème.

2.3.4 La conduite des raisonnements

Les stratégies que nous avons décrites pour les listes demeurent valables pour les ensembles et sont utilisées. Nous organisons de nouveau notre raisonnement en étapes et nous utilisons pour cela la structuration mise en place sur l'univers. Une étape a pour objectif de trouver une valeur pour un ou des attributs ; selon le résultat obtenu, le système choisit de poursuivre par l'une ou l'autre des étapes indiquées.

Dans le cas des listes, l'étape terminale consistait à donner une valeur à l'attribut implantation. Pour les ensembles, il nous faut définir autrement l'étape terminale ; on peut choisir :

- Une étape terminale donnant la valeur "souhaité" à l'une des représentations possibles. Cette solution a été utilisée pour l'implantation de MOTS-CLES décrite ci-après. Si au cours de son raisonnement, le système a trouvé des valeurs pour d'autres représentations, on les obtiendra aussi en consultant la valeur des attributs correspondants.

- Une étape terminale imposant de valuer certaines représentations particulières. Ce choix peut être retenu si l'utilisateur veut prendre cette représentation et a envie de savoir "ce qu'en pense" le système.

- Une étape terminale imposant de valuer toutes les représentations . Cette solution est intéressante lorsqu'on a plusieurs objets à représenter et qu'on veut faire des choix acceptables pour chacun d'eux tout en respectant des contraintes que le système ne prend pas en compte lui-même, par exemple un même représentation pour des objets intervenant comme arguments d'une opération sur les ensembles.

D'autres stratégies sont évidemment envisageables ; celles-ci nous ont paru

suffisamment représentatives de la diversité des objectifs possibles. Comme pour les listes, nous donnons en annexe la base de règles sur les ensembles actuellement implantée dans SAIDA et présentons un dialogue possible en utilisant cette base pour un choix d'implantation de l'objet MOTS-CLES.

Exemple de mise en oeuvre :

Le choix d'une étape terminale caractérisée par l'affectation de la valeur "souhaité" à l'un des attributs représentant des implantations conduit au dialogue suivant pour l'objet MOTS-CLES de la procédure COMPTE_MOTS.

Etape issu de : le but de cette étape est de savoir si l'objet est lié à d'autres dans certaines opérations et de pouvoir attirer l'attention de l'utilisateur sur ce point en fin de session.

S : Avons-nous - l'ensemble étudié est le résultat d'une recopie - ?
U : Non
S : Avons-nous - l'ensemble étudié est le résultat d'une opération ensembliste - ?
U : Non

Etape fichier : le but est de savoir si on doit utiliser un fichier.

S : Avons-nous - le support est imposé à fichier à accès direct - ?
U : Non
S : Avons-nous - la taille de l'ensemble est trop importante pour être traitée en mémoire centrale - ?
U : Non

Etape tableau de booléens : le but est de savoir si on peut utiliser une implantation par tableaux de booléens.

S : Quelle est la valeur de - le type de l'élément - ?
U : Chaîne de caractères
S : Avons nous - les éléments du référentiel sont connus pour chaque exécution - ?
U : Oui
S : Avons-nous - type élément peut servir d'index dans un tableau en ADA - ?
U : Non
S : Avons-nous - il existe des opérations ensemblistes avec l'objet comme composant - ?
U : Non

S : Avons-nous - le référentiel de l'ensemble est partagé par d'autres ensembles - ?
U : Non
S : Avons-nous - la taille de l'ensemble est égale à celle du référentiel - ?
U : Oui

Etape tableau de valeurs : le but est d'essayer de parvenir à une représentation par tableau de valeurs.

S : Avons-nous - beaucoup d'adjonctions/suppressions en position quelconque - ?
U : Non
S : Avons-nous - la taille de l'ensemble varie peu en cours d'exécution - ?
U : Oui
S : Avons-nous - il existe un ordre sur les éléments - ?
U ; Oui
S : Avons-nous - on souhaite favoriser les opérations APPARTIENT, MIN, MAX - ?
U : Oui

Etape finale : le système livre ses conclusions.

Tableau de booléens indexé par le type des éléments de l'ensemble = refusé
Tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément = refusé
Tableau trié de valeurs des éléments = souhaité
Tableau ordre quelconque de valeurs des éléments = valeur non connue
Liste chaînée de valeurs dans un ordre quelconque = valeur non connue
Liste chaînée triée de valeurs = valeur non connue
Arbre binaire de recherche = valeur non connue
Sous-listes chaînées avec répartition par fonction de hachage = valeur non connue
Fichier à accès direct = refusé.

Les interrogations et déductions du système se sont arrêtées dès que l'un des attributs relatifs à une implantation a pris la valeur souhaité; il s'agit dans cet exemple de tableau trié de valeurs. On constate qu'au cours de la résolution du problème trois autres attributs ont reçu une valeur, les autres n'ont pas été valués, ce qui est noté valeur non connue dans la base de faits.

2.4 ACCROITRE L'EXPERTISE DU DOMAINE ET ORGANISER LE PROCESSUS DE CONSTRUCTION

Au terme de cette description, nous voudrions tirer quelques leçons de l'expérience acquise et élargir notre horizon dans deux directions. Le caractère nuancé et peu formalisé des raisonnements dans notre domaine d'application nous a obligés à bien délimiter dans un premier temps la catégorie de problèmes que la base de connaissances construite peut aider à résoudre, mais nous avons évidemment envie de pouvoir l'étendre ; nous donnons donc rapidement un aperçu des directions dans lesquelles un tel travail pourrait rapidement être entrepris. Par ailleurs, nous avons décrit une conception de base de connaissances dans le domaine de la programmation, nous avons l'expérience de travaux analogues dans d'autres domaines ; nous tentons donc de résumer ce qui nous paraît faire partie d'une démarche commune de conception de tels systèmes ainsi que les difficultés plusieurs fois rencontrées sur lesquelles devraient porter de nouvelles recherches.

2.4.1 Accroître l'expertise du domaine

Davantage de types, de représentations et des critères plus fins

Une première extension envisageable consiste à enrichir la bibliothèque de types et à doter la base de connaissances des règles de choix de représentation correspondantes.

Pour les types existants, on peut d'une part affiner les critères de choix proposés afin de prendre en compte de plus en plus de cas particuliers rencontrés dans les applications ; le système ne détecte pas, dans sa version actuelle, que l'ensemble d'épreuves du problème de Hoare (partie I, chapitre 1) peut être géré comme une pile et il ne dispose pas de cette implantation pour les ensembles. On peut d'ailleurs ajouter de nouvelles représentations, notamment pour les ensembles, certaines représentations par des arbres favorisant l'implantation de groupes d'opérations particuliers.

On peut ensuite évidemment augmenter le nombre de types pris en compte dans la base, et ce travail est prévu dans le cadre du contrat dont la réalisation de SAIDA fait l'objet.

Etude globale de l'implantation des objets d'un algorithme

Une autre extension plus intéressante à notre sens consisterait à accroître le niveau d'expertise du système pour lui permettre d'étudier globalement l'implantation de tous les objets d'une application, au lieu de le faire objet après objet, sans tenir compte des études et des propositions déjà faites pour d'autres objets. Cela suppose une étude la nature des liens

qui unissent les différentes informations d'une application, plus exactement de ceux de ces liens qui doivent être pris en compte lors d'un problème de choix d'implantation en mémoire.

A titre d'exemples, nous avons repéré dans les exercices traités :

- Des liens de simple voisinage, par exemple, une table TPRIX donnant des prix à partir de codes d'articles, et une liste LCLIENTS de "bons clients" dans un problème de facturation ; TPRIX et LCLIENTS n'interviennent pas dans la même opération ; ils sont utilisés à deux étapes différentes de l'algorithme de calcul ; ils sont "voisins". Ces objets, qualifiés de voisins, peuvent être de même type ou, comme ici, de types différents ; la représentation de l'un n'influe pas sur la représentation de l'autre.

- Des liens "sémantiques", par exemple, dans le problème du Hit-Parade, on sait qu'on remplit chacune des listes résultats à partir de la liste des données, et on a donc une relation indiquant que la somme des nombres d'éléments des listes résultats est égale au nombre d'éléments de la liste donnée. De telles relations caractéristiques des problèmes devraient être fournies comme "invariants" avec les textes d'algorithmes.

- Des liens indiquant l'appartenance à la liste des paramètres d'une même procédure, par exemple UNION (E1, E2, E3) qui réalise dans E3 la réunion de E1 et E2.

- Des liens traduisant traduisant le fait qu'un objet est composant d'un autre, par exemple, la liste des ensembles de sommets formant les composantes connexes d'un graphe ; il faudrait dans ce cas savoir exprimer des relations entre la représentation du tout et celle des parties.

Il faudrait ensuite exprimer les contraintes créées par ces différents liens, et probablement bien d'autres, sur les implantations des objets considérés.

Acquisition automatique de certaines informations

Toute l'acquisition des connaissances repose actuellement sur un dialogue avec l'utilisateur ; pourtant, le système peut accéder, comme l'utilisateur, au texte de la procédure contenant l'objet à implanter et en extraire certaines informations ; il faut essayer de déterminer lesquelles et comment.

Pour apporter quelques éléments de réponse à cette question, nous allons reprendre, rubrique par rubrique, les informations utilisées dans le système, et nous demander, pour chacune d'elles, si elle pourrait être obtenue automatiquement par analyse du texte de la procédure ADA correspondante.

- Les contraintes externes :

Pour renseigner cette rubrique, aucune information ne figure dans un texte de procédure. Il faudrait envisager d'adjoindre au texte des spécifications décrivant ces contraintes, c'est la solution choisie dans le système LIBRA.

Sinon, le dialogue avec l'utilisateur demeure la seule solution. Notons que ces informations sont indispensables pour effectuer des choix d'implantations et qu'on ne dispose pas de moyens pour les exprimer de façon un peu systématique.

- Les caractéristiques des objets :

A propos de l'objet dont l'utilisateur a fourni le nom et le type, on recueille des informations de natures différentes.

Certaines sont purement statiques, par exemple le type des éléments, la borne supérieure du nombre d'éléments, l'existence d'une relation d'ordre sur les éléments, etc... Le type des éléments figure dans les déclarations de la procédure, les deux autres caractéristiques ne peuvent être obtenues systématiquement dans le texte.

Certaines sont liées à une dynamique d'exécution, mais demeurent indépendantes des données, par exemple la taille de l'objet peut rester "globalement fixe" si chaque adjonction est compensée par une suppression.

Par contre cette taille de l'objet peut dépendre des données fournies au programme.

- Les caractéristiques des traitements

A propos des opérations utilisées sur l'objet à implanter, on cherche aussi des renseignements de natures différentes : on distingue, par exemple, entre l'existence d'une opération (existe-t-il des adjonctions ?), la fréquence d'une opération (existe-t-il beaucoup d'adjonctions), la fréquence relative d'une opération par rapport à d'autres (y a-t-il plus d'adjonctions/suppressions que de consultations ?), l'ordre d'exécution des opérations les unes par rapport aux autres.

Parmi toutes ces caractéristiques, celles qui sont purement "syntaxiques" sont aisément identifiables par analyse du texte. D'autres, comme celles liées aux données ne s'y trouveront jamais. Entre les deux, des mécanismes fins d'analyse d'algorithmes, notamment des modèles probabilistes dynamiques développés pour des études de complexité [FLA,80], [FRA,88] pourraient apporter des informations.

Les auteurs de SETL ont développé un système d'analyse de programme SETL [SCHON,81] afin de déterminer certaines caractéristiques pour optimiser la représentation

des ensembles à l'issue de la phase de compilation. Les ensembles ont en SETL une représentation standard que l'on peut assimiler à celle que nous nommons sous-listes chaînées avec répartition par fonction de hachage. L'optimisation proposée concerne les ensembles ayant un référentiel commun et l'opération d'appartenance sur ces ensembles. Leur algorithme détermine par parcours du texte les objets qui appartiennent à plusieurs ensembles et en donne une implantation n'obligeant pas à recalculer la fonction de hachage à chaque consultation de l'objet.

Dans LIBRA, on trouve une description des contraintes en place mémoire et temps d'exécution à prendre en compte ainsi que des renseignements sur les tailles d'objets et la fréquence de certaines opérations ; il n'y a aucun dialogue prévu avec l'utilisateur.

Dans SAIDA, on pourrait extraire par analyse syntaxique du texte de procédure certaines des informations mentionnées ci-dessus, en particulier pour chaque objet, la liste de opérations utilisées sur cet objet, ainsi que le niveau d'imbrication des appels de chacune d'elles. Une telle analyse permettrait de renseigner les rubriques correspondantes de la base de faits et donc d'alléger le dialogue avec l'utilisateur. Un exemple de mise en oeuvre d'une telle analyse est fourni en quatrième partie.

2.4.2 Organiser le processus de construction

Nous avons décrit dans les paragraphes précédents certains aspects du processus de construction de la base de connaissances de SAIDA. Les ouvrages de synthèse traitant de la construction de systèmes experts indiquent des phases de construction à respecter [BON,86], [WATERM,86] depuis l'étude d'opportunité jusqu'à la mise en oeuvre du système, mais ils ne fournissent pas de guide pour un recueil et une représentation méthodiques des connaissances.

Le formalisme que nous utilisons, ensemble d'attributs et règles de production, ne fournit pas a priori de support pour un recueil organisé et structuré des informations. Nous avons tenté d'y remédier en faisant apparaître la structuration d'univers que nous avons retenue et en classant les concepts utilisés à partir de cette structuration ; cela nous permet de présenter les règles triées selon les intermédiaires d'implantation vers lesquelles elles permettent de s'orienter.

Mais, ces modes de structuration sont insuffisants, il faudrait par exemple pouvoir aussi classer les règles selon d'autres critères comme des caractéristiques utilisées en prémisses. Il existe actuellement des outils de développement de systèmes experts beaucoup plus performants, le cadre de description de générateur d'application décrit dans

[LAURE,88] montre tout ce qui est a priori possible. Ils sont pour un temps encore réservés à des spécialistes, tant par les machines qui les supportent que par la compétence et l'expérience que leur utilisateur doit posséder. On peut citer dans cette catégorie KEE, ART, KNOWLEDGE CRAFT, [LAURE,87], KOOL [BUL,88] qui permettent de structurer dans un environnement orienté objet des concepts et des règles, et de les structurer selon différents points de vue en utilisant notamment des mécanismes d'héritage multiple.

Cependant, certains logiciels de développement sont ou vont être commercialisés sur compatibles PC ; ils deviennent ainsi plus facilement accessibles à une plus large catégorie d'utilisateurs, rendant encore plus aiguë la question des méthodes de construction de systèmes à bases de connaissances.

Nous dégageons dans la suite de ce paragraphe des points qui nous ont semblé importants dans le processus de construction de la base avec pour objectif de travailler "sur le papier" comme si les outils d'expression et de structuration nécessaires existaient et de choisir ou créer chaque fois que possible des outils permettant ce mode de travail.

Dégager et organiser les concepts du domaine

Nous avons d'abord recherché les caractéristiques des algorithmes qu'il fallait prendre en compte pour résoudre un problème de représentation d'objet abstrait. Par rapport à l'ensemble des concepts nécessaires dans la base de connaissances, nous parlerons de concepts liés à la description du problème à résoudre. La difficulté consiste là à ne prendre en compte que des caractéristiques utiles à la résolution et donc de réaliser une bonne abstraction de l'univers réel, puis à se donner les moyens d'organiser peu à peu les concepts ainsi mis en évidence.

Nous avons également décrit l'univers des implantations auxquelles nous voulions aboutir. Nous avons enfin défini des concepts intermédiaires pour exprimer plus facilement des raisonnements élémentaires. Alors que les concepts liés à la description du problème et ceux décrivant les représentations se trouvent dans les manuels d'informatique, ceux qui ont servi de support à l'expression de raisonnements étaient à inventer.

On retrouve cette démarche dans d'autres applications. Pour le diagnostic de pannes [GRAN,86a], [GRAN,87], on retrouve la description des caractéristiques physiques de l'installation, la description des pannes possibles et il reste à jalonner le chemin conduisant de certaines caractéristiques à certaines pannes. On cherche aussi à structurer l'univers de travail, et on a naturellement besoin de le structurer selon différents points de vue : la

proximité géographique des éléments, la fonction à l'accomplissement de laquelle ils concourent dans l'installation, le type de panne (mécanique ou électrique) dans laquelle ils peuvent intervenir, etc... . On utilise des concepts intermédiaires qui sont ceux des experts du domaine comme les notions d'indice de panne, de contexte de fonctionnement, de déviance par rapport à une norme, etc... .

Retenons qu'il faut disposer de moyens d'expression permettant de décrire des concepts, de les structurer à différents niveaux et selon différents points de vue. Cela est important à la fois pour la phase de création de la base que nous venons d'envisager, pour son utilisation, notamment pédagogique à laquelle nous consacrons le chapitre suivant et pour la phase de maintenance que nous n'abordons pas ici.

Exprimer des raisonnements

Les raisonnements élémentaires sont exprimés par des règles. Il faut également pouvoir les organiser. Des modes d'organisation automatique des règles ont été proposés dans les moteurs existants parce qu'une organisation judicieuse peut accélérer la résolution des problèmes en diminuant à chaque étape du raisonnement le nombre de règles que le système essaie d'appliquer.

On trouve ainsi des classements selon les conclusions, selon des caractères des membres gauches, selon la fréquence d'utilisation dans le système ou encore selon des métarègles définies par le concepteur de l'application. Il faudrait étudier de quels types de classement on aurait besoin pour la création de la base.

Organiser les règles, les réorganiser au cours de la résolution d'un problème, c'est exprimer des raisonnements sur les raisonnements ; dès qu'une application devient un peu importante, il est nécessaire d'exprimer des raisonnements de ce type ; il faut donc veiller à ce que l'outil utilisé le permette.

Chapitre 3

LA BASE DE CONNAISSANCES DE SAIDA ASPECTS PEDAGOGIQUES

SAIDA, l'environnement de création de programmes que nous construisons, est un environnement d'enseignement. La base de connaissances que nous venons de décrire au chapitre précédent doit servir dans ce système à enseigner à l'étudiant les raisonnements à faire quand il choisit une implantation pour un objet abstrait de son algorithme.

Une base de connaissances et le moteur d'inférences qui permet de l'utiliser pour résoudre des problèmes peuvent être le noyau de nombreuses réalisations pédagogiques. Nous rappelons dans le premier paragraphe les principaux modes d'utilisation rencontrés dans les environnements pédagogiques comportant des bases de connaissances.

Nous indiquons ensuite les modes d'utilisation que nous retenons dans le système SAIDA. Nous y expliquons notamment notre choix de ne pas avoir de modèle de l'élève et de laisser à l'enseignant certaines décisions concernant l'adaptation du système à une population d'élèves donnée.

Pour chaque mode d'utilisation, nous indiquons à quelles techniques des systèmes à base de connaissances nous pouvons faire appel et nous illustrons ce mode par des exemples de dialogues obtenus dans SAIDA.

3.1 SYSTEMES EXPERTS DANS L'ENSEIGNEMENT : EXPERTISE DU DOMAINE, ADAPTATION PEDAGOGIQUE

G. CLAES rappelle dans [CLAE,88] que la contribution des techniques d'intelligence artificielle dans les systèmes d'enseignement assisté par ordinateur concerne essentiellement la représentation des connaissances, la résolution des exercices et la qualité des interfaces entre le système et l'élève. Nous envisageons dans ce chapitre surtout les deux premiers points.

L'activité d'enseignement pose en effet la question de la transmission des contenus dans une discipline donnée, mais aussi celle de la transmission de stratégies permettant d'utiliser ces contenus. Les systèmes à base de connaissances possèdent ce savoir et le "savoir utiliser" qui lui est associé, c'est l'expertise du domaine ; nous faisons l'hypothèse qu'ils peuvent aider l'élève à se l'approprier s'ils sont utilisés dans des cadres adaptés aux besoins des élèves, c'est à dire s'ils prennent en compte aussi une expertise pédagogique.

Ce volet pédagogique des systèmes à base de connaissances peut prendre des formes diverses ; nous indiquons les principales.

3.1.1 Utilisation sans adaptation pédagogique, limites

Des systèmes à base de connaissances comme celui décrit au chapitre précédent peuvent être utilisés avec des élèves. Par rapport à des systèmes d'enseignement assisté par ordinateur classiques, ils ont l'avantage de ne pas reposer sur une banque d'exercices prédéfinis, et donc de résoudre n'importe lequel des problèmes soumis par l'apprenant dans les limites du champ de compétence du système ; ils offrent également par leur mécanisme de trace la possibilité d'étudier pas à pas le raisonnement effectué par le système ; on peut encore leur demander de fournir plusieurs solutions lorque le problème s'y prête et d'exhiber plusieurs raisonnements conduisant à une solution.

Dès qu'on dépasse la simple résolution du problème posé, les limites de systèmes non conçus pour l'enseignement apparaissent très vite :

- **Les traces du raisonnement** sont intéressantes à condition que ce raisonnement ait été conduit de la façon dont l'élève aurait pu le conduire ; la trace fournie par un interprète PROLOG n'est pas très éclairante à cet égard. Il est important aussi que ce raisonnement soit structuré, la lecture d'une liste de 50 règles appliquées ne peut pas non plus tenir lieu d'explication et de guide de raisonnement. Dans les exemples fournis au chapitre précédent, nous avons organisé l'application des règles de la base en étapes pour aboutir à un dialogue acceptable par l'utilisateur .

Il faut enfin veiller à ce que le libellé des règles soit compris par l'élève. Or, dans les systèmes à règles de production, on peut se heurter à deux problèmes. D'une part on a tendance à utiliser des noms de prémisses courts mais peu explicites, et nous n'y avons pas échappé en prenant représentation1 et représentation2 comme intermédiaires dans la conduite du choix d'implantation d'un objet de type liste. D'autre part, la notation " si P alors C " exprime entre les prémisses P et les conclusions C des liens de natures différentes qu'il faudrait expliciter . Enfin, les prémisses n'ont pas toutes le même rôle, souvent l'une d'elles est importante pour le raisonnement, les autres ne représentent qu'un contexte d'utilisation de la règle.

- La possibilité de **résoudre un problème en faisant des raisonnements différents** est certes intéressante à montrer, mais elle est mieux utilisée dans un suivi pas à pas du travail de l'élève. Dans un tel suivi, le système "observe" le travail de l'élève au cours de la résolution d'un problème et regarde si l'une des méthodes qu'il connait conduit aux mêmes résultats intermédiaires.

- La **formalisation de la connaissance,** notamment son morcellement en un grand nombre d'attributs et de règles, peut s'avérer peu éclairante pour l'apprenant si le domaine de travail n'a pas été suffisamment structuré ou si cette structuration n'est pas apparente

Clancey est l'un des premiers à construire un système d'enseignement, GUIDON, ayant pour module expert la base de connaissances en médecine (450 règles) et le moteur de MYCIN. Il souligne dans [CLAN,82] les insuffisances de ce type d'approche malgré l'adjonction de règles tutorielles pour gérer le dialogue avec l'élève et exploiter les renseignements fournis par MYCIN à propos d'étude de cas soumise à l'étudiant. Il souligne notamment que MYCIN n'a pas de connaissances en planification ni en structuration de son raisonnement, ce qui lui semble nécessaire pour une utilisation à des fins pédagogiques. Cette expérience l'a conduit à développer NEOMYCIN [CLAN,81], un nouveau système pour l'enseignement du diagnostic médical qui utilise des connaissances analogues à celle de MYCIN sur le plan du contenu, mais totalement restructurées sur le plan de la forme, ainsi que d'autres informations relatives au domaine et à la pédagogie.

En supposant que les bases de connaissances actuelles soient mieux structurées que celle de MYCIN, nous examinons maintenant la fonction des modules créés pour des besoins pédagogiques.

3.1.2 La conduite du dialogue

Dans un système expert, le dialogue avec l'utilisateur se limite en général à l'acquisition d'informations auprès de ce dernier, et à des explications à la demande, si les questions posées le surprennent. En fin de résolution, une trace plus ou moins détaillée et commentée explique le résultat obtenu.

Dans un système à vocation pédagogique, le dialogue doit guider l'élève dans sa démarche de résolution de problème, lui fournir de l'aide, expliquer la stratégie développée par le système. Ce dialogue englobe celui qui peut être conduit par le système expert et le dépasse largement.

Un premier module souvent ajouté à un système à base de connaissances est donc un module de gestion pédagogique du dialogue avec l'élève. C'est le rôle des 220 règles "tutorielles" construites pour GUIDON autour du système MYCIN. Ce dialogue s'appuie souvent sur des connaissances pédagogiques concernant la matière enseignée et sur un modèle de l'élève. Il fait une place de choix aux explications. Ces différents aspects peuvent être imbriqués dans ce module de dialogue, mais on tend aujourd'hui à les étudier séparément [SAF,85], [VIV,87a], [PAL,88].

3.1.3 L'expertise pédagogique dans la matière enseignée

Cette expertise de l'enseignant se manifeste dans l'organisation de la matière à enseigner, dans le diagnostic des erreurs commises par l'apprenant, dans l'organisation de la conduite d'une session en fonction des résultats obtenus.

- L'ordre de présentation des concepts

Si certaines notions, ou certains modes de résolution n'ont pas été étudiés par l'élève, il faut que le système ne les propose pas. VIVET dans [VIV,87] illustre cette idée dans la métarègle suivante issue de son système CAMELIA [VIV,84] :

si l'objectif du problème est factoriser
et les nombres complexes n'ont pas encore été étudiés
alors supprimer de la liste des plans possibles les plans utilisant les complexes.

- Le diagnostic des erreurs

Lors du suivi pas à pas des réponses de l'élève, le système s'assure du fait qu'il sait conduire un raisonnement produisant cette réponse. Lorsque ce n'est pas le cas, la réponse est supposée erronée. Certains systèmes contiennent des règles modélisant les raisonnements erronés les plus couramment rencontrés ; de telles connaissances permettent de donner des explications beaucoup plus fines que le simple constat "ce résultat est faux" ... "une réponse juste est ... ."

- La conduite de la session

La conduite de la session comporte le choix du contenu du dialogue à un instant donné, l'enchaînement des dialogues et souvent l'enchaînement des activités proposées aux apprenants. Dans le cas d'une réponse erronée, même si le système ne sait pas identifier le "raisonnement" qui a conduit à cette réponse, il peut reconnaître une catégorie d'erreurs et provoquer l'impression de messages adaptés ou même modifier le déroulement prévu et proposer des révisions.

Supposons qu'on obtienne par exemple la réponse 53 m^2 au lieu de 53 cm^2. On peut diagnostiquer une erreur sur la partie "unité" du résultat et infléchir le déroulement de la session pour en tenir compte.

- Les explications

Une part du dialogue consiste à fournir des explications sur le raisonnement . Ces explications peuvent concerner un apport de connaissances manquantes, les raisons qui conduisent à appliquer une règle, la stratégie globale de la résolution. L'insuffisance des traces de déclenchement de règles ayant été maintes fois remarquée, Vivet propose dans [VIV,87] le classement possible suivant pour les explications :

COMMENTER	simple remise en forme de la trace du déclenchement des règles
EXPLIQUER	préciser le pourquoi de l'existence d'une règle (rappel sur la connaissance ou sur le pourquoi de son application.)
JUSTIFIER	relatif à la stratégie de raisonnement, aux choix effectués.

D'autres auteurs [SAF,85] s'attachent à répondre à des questions de la forme "pourquoi pas".

Dans tous les travaux récents décrivant des systèmes à base de connaissances pour l'enseignement, ces connaissances pédagogiques sont données sous forme de règles séparées dans des système experts coopérants [VIV,87b] ou des métarègles [PAL,88]. Elles renvoient plus ou moins explicitement à un modèle de l'élève.

3.1.4 Le modèle de l'élève

Ce modèle reflète des hypothèses en matière d'apprentissage notamment. Il peut ne concerner que la discipline enseignée, le système mémorise ce que l'élève sait, ce qu'il ne sait pas encore, les règles qu'il applique le plus souvent, celles qu'il n'applique pas... Il peut comprendre des données pré-enregistrées sur l'apprenant ou se fabriquer lui-même son image de l'élève. Il est toujours complété en cours de session par l'observation du comportement de l'élève et utilisé pour la conduite du dialogue.

Nous venons de donner un aperçu des directions dans lesquelles des travaux récents ont été développés pour construire un environnement d'enseignement autour d'un système à base de connaissances. Nous consacrons le paragraphe suivant aux choix que nous avons faits dans SAIDA pour une utilisation pédagogique de la base de connaissances précédemment décrite.

3.2 COMPLETER ET UTILISER LA BASE DE CONNAISSANCES SUR LES REPRESENTATIONS D'OBJETS A DES FINS PEDAGOGIQUES

Notre objectif est d'utiliser la base de connaissances construite au chapitre précédent pour transmettre une expertise, un savoir-faire, dans le domaine du choix d'implantation de données abstraites. Le savoir, c'est à dire les différentes représentations possibles pour un type, les algorithmes implantant de façon efficace certaines opérations, les critères généraux de choix se trouvent dans les manuels, peut être consulté grâce à la bibliothèque de types et fait l'objet d'un cours.

Avec le système, l'apprenant doit acquérir un savoir-faire, c'est à dire la capacité de choisir une bonne représentation pour un objet abstrait de l'algorithme qu'il construit en utilisant à bon escient les connaissances disponibles.

Cependant, contrairement à l'expert, l'apprenant n'est pas encore un familier du domaine. Pour le devenir, il doit savoir organiser ce domaine en concepts abstraits à partir desquels il construira des raisonnements et résoudra des problèmes. Il faut donc d'une part lui permettre de s'approprier les abstractions du domaine sur lesquelles il peut construire des raisonnements et, d'autre part, l'aider à chaque pas de la résolution d'un problème.

3.2.1 S'approprier les abstractions du domaine pour conduire des raisonnements

Nous faisons l'hypothèse que le travail avec un système à base de connaissances peut aider l'élève à acquérir et structurer les concepts du domaine.

- Simple lecture des règles

La lecture des règles constitue une première forme de familiarisation avec les différents concepts manipulés par le système. L'utilisation des règles avec cet objectif suppose l'existence d'un bon lexique de présentation du sens attaché à chaque attribut de la base de règles.

- Stratégies de raisonnement prenant pour base cette structuration

Le système doit entraîner l'élève à conduire des raisonnements s'appuyant sur les abstractions du domaines jugées importantes par l'enseignant. Nous avons vu au chapitre précédent qu'il y avait plusieurs façons de structurer le raisonnement effectué lors d'un choix d'implantation ; certains de ces modes de structuration correspondent à des stratégies bien

identifiées (dirigée par les données, par les résultats, etc...).

La composante pédagogique à adjoindre aux règles du système est alors une structuration du raisonnement en étapes, selon la stratégie choisie, mettant en évidence les concepts intermédiaires jugés importants.

- Exemple de mise en oeuvre : choix d'une implantation d'ensemble dirigé par les données

L'accent doit être mis sur les données du problème. Nous devons donc mettre en évidence les concepts qui permettent l'organisation de l'univers de description du problème. Dans cet univers, nous avons mis en évidence trois groupes de caractéristiques : les contraintes externes, les informations relatives à l'objet à implanter, les informations relatives aux traitements.

L'attention de l'élève peut être attirée sur le mode de structuration des caractéristiques du problème à résoudre par un recueil d'informations organisé de cette façon et explicitement apparent.

Réalisation : La stratégie de raisonnement du système est la suivante :

1 - Recueillir toutes les informations relatives aux contraintes (c'est à dire trouver une valeur pour tous les attributs de cette catégorie)
2 - Recueillir toutes les informations relatives aux objets
3 - Recueillir toutes les informations relatives aux traitements
4 - Conclure avec l'une des stratégies proposées précédemment, c'est à dire donner une valeur à l'une des implantations d'ensembles, ou à toutes.

Elle s'exprime par une organisation de l'application des règles en trois étapes ayant chacune pour objectifs des acquisitions de données.

Le dialogue qui résulte de cette stratégie peut s'avérer lourd, ce n'est certainement pas celui qu'on trouve dans un système expert pour professionnel, mais c'est une forme de dialogue que l'enseignant peut souhaiter pour entraîner à une analyse méthodique et complète du problème.

Au début de l'exécution de chaque étape, le système indique l'objectif de l'étape ; à l'issue de l'étape, il peut indiquer tout ce que les informations acquises lui ont permis de déduire. Il faut noter que dans ce cas, l'ensemble des informations souhaitées sera recueilli même si le système peut proposer une représentation avant la fin de ce processus de recueil.

Description des étapes créées pour implanter cette stratégie

Une étape oblige le système à appliquer ses règles dans un certain ordre. Cet ordre est défini par les attributs pour lesquels on veut obtenir une valeur. La valeur d'un attribut peut être obtenue en posant directement une question à l'utilisateur ou en appliquant des règles susceptibles de conduire à la valuation recherchée. Dans l'exemple de recueil systématique d'informations que nous décrivons, nous n'organisons que les questions à poser à l'élève. Nous donnons ci-dessous pour chaque étape son objectif et la liste des attributs à valuer.

Etape et/contraintes :

Cette étape a pour objectif de faire décrire les contraintes externes à l'algorithme qui doivent être prises en compte dans une étude d'implantation d'objet abstrait.

Liste des attributs:

le support est imposé à fichier à accès direct
on souhaite gérer au mieux l'espace mémoire
des problèmes de place en mémoire centrale
il y a nécessité d'optimiser les temps d'accès

Etape et/objets :

Cette étape a pour objectif de faire décrire les caractéristiques de l'objet à implanter et ses interactions éventuelles avec d'autres objets.

Liste des attributs:

le type de l'élément
les variations du nombre d'éléments
les éléments du référentiel sont connus pour chaque exécution
il existe un ordre sur les éléments
il existe une bonne fonction de hachage
l'ensemble étudié est le résultat d'une recopie
l'ensemble étudié est le résultat d'une opération ensembliste
il existe une borne connue de la taille de l'ensemble
la taille de l'ensemble est voisine de celle du référentiel
la taille de l'ensemble est en permanence plus petite que celle du référentiel
la taille de l'ensemble est trop importante pour être traitée en mémoire centrale

Etape et/procédures

L'objectif de cette étape est de faire acquérir des informations relatives aux opérations effectuées dans l'algorithme sur l'objet à implanter.

Liste des attributs:

il existe des opérations ensemblistes avec l'objet comme composant

il y a beaucoup d'adjonctions
beaucoup d'adjonctions/suppressions en position quelconque
on souhaite favoriser les tests d'app/min/max

Dialogue obtenu pour l'implantation de la procédure COMPTE_MOTS avec ces étapes.

Nous donnons ci-dessous un exemple d'appel au système à base de connaissances muni des étapes que nous venons de décrire. On y retrouve au niveau du dialogue les questions prévues, nous montrons les informations intermédiaires qui sont déduites et l'état de la base de faits à l'issue de la session.

S : *On étudie successivement les contraintes, les objets, les procédures.*
S : *Etape et/contraintes : On étudie les contraintes.*
S : Avons-nous - le support est imposé à fichier accès direct - ?
U : non
S : Avons-nous - on souhaite gérer au mieux l'espace mémoire - ?
U : oui
S : Avons-nous - des problèmes de place en mémoire centrale - ?
U : non
S : Avons-nous - il y a nécessité d'optimiser les temps d'accès - ?
U : oui

A l'issue de cette étape, on trouve dans la base de faits:
le support est imposé à fichier accès direct = faux
on souhaite gérer au mieux l'espace mémoire = vrai
des problèmes de place en mémoire centrale = faux
il y a nécessité d'optimiser les temps d'accès = vrai

Aucun autre attribut relatif à la description du problème n'est valué parce qu'aucune règle ne devient applicable avec les renseignements fournis.

S : *Etape et/objets : on étudie les objets.*
S : Quelle est la valeur de - le type de l'élément - ?
U : chaîne de caractères
S : Quelle est la valeur de - les variations du nombre d'éléments - ?
U : fixe
S : Avons-nous - les éléments du référentiel sont connus pour chaque exécution - ?
U : oui
S : Avons-nous - il existe un ordre sur les éléments - ?
U : oui

S : Avons-nous - il existe une bonne fonction de hachage - ?
U : je ne sais pas
S : Avons-nous - l'ensemble étudié est le résultat d'une recopie - ?
U : non
S : Avons-nous - l'ensemble étudié est le résultat d'une opération ensembliste - ?
U : non
S : Avons-nous - il existe une borne connue de la taille de l'ensemble - ?
U : oui
S : Avons-nous - la taille de l'ensemble est voisine de celle du référentiel - ?
U : je ne sais pas
S : Avons-nous - taille ensemble est en permanence bien plus petite que celle du référentiel - ?
U : non
S : Avons-nous - la taille de l'ensemble est trop importante pour être traitée en mémoire centrale - ?
U : non

A l'issue de cette étape, comme dans le cas précédent, la base de faits contient les couples (attribut, valeur) pour lesquels nous venons de fournir des réponses ; elle contient de plus le résultat des déductions qui ont été effectuées.

éléments homogènes = vrai
arbre binaire de recherche = refusé
liste chaînée avec hachage = valeur non connue
conseil1 (sert à afficher un conseil à l'utilisateur) = valeur non connue
fichier à accès direct = refusé
tableau de valeurs des éléments = souhaité
tableau ordre quelconque de valeurs des éléments = valeur non connue

S : *Etape et/procédures : on étudie les opérations.*
S : Avons-nous - il existe des opérations ensemblistes avec l'objet comme composant - ?
U : non
S : Avons-nous - il y a beaucoup d'adjonctions - ?
U : non
S : Avons-nous - beaucoup d'adjonctions/suppressions en position quelconque - ?
U : non
S : Avons-nous - on souhaite favoriser les tests d'app/min/max - ?
U : oui

A l'issue de cette étape, en plus des valeurs fournies par dialogue, on obtient :
vers-chaînée = valeur non connue
organisation = triée

tableau trié de valeurs des éléments = souhaité
liste chaînée de valeurs dans un ordre quelconque = valeur non connue

La structuration des représentations peut être mise en évidence par un dialogue qui explique vers quels types de représentations on veut s'orienter et justifie ainsi les questions posées.

Réalisation : la conduite des inférences sera cette fois assurée par une succession d'étapes proposant les objectifs suivants :

- voir s'il faut utiliser un fichier
- voir si on peut utiliser un tableau de booléens
- voir si on peut utiliser un tableau de valeurs
- voir si on peut utiliser une représentation chaînée
  * par technique de hachage
  * avec un arbre binaire
  * par liste chaînée

En obligeant le système à passer par chacune de ces étapes, on montre à l'élève comment envisager de façon organisée tous les cas possibles d'implantation. Cette stratégie est utilisée dans l'exemple développé au paragraphe suivant.

Ces deux exemples montrent qu'on peut mettre en évidence des structurations d'univers en structurant le dialogue proposé à l'élève. Ce dialogue est lié à l'univers mais aussi au raisonnement. Nous nous attachons maintenant à montrer comment mettre en évidence la structure du raisonnement.

3.2.2 Les raisonnements du domaine

Les deux exemples que nous venons de donner pour la structuration de l'univers illustrent aussi des formes de raisonnements. Il est possible d'entraîner l'élève à des raisonnements en mettant en évidence la structure de ce raisonnement. A coté des deux exemples précédents, on peut montrer des démarches moins "systématiques" au sens du domaine de travail mais plus efficaces parce que conduisant plus rapidement à la solution dans beaucoup de cas. Ce serait plutôt la stratégie de l'expert, poser quelques questions pertinentes et aller rapidement à certaines solutions, éliminer rapidement certaines autres. Une stratégie de ce type nommée "et/vite" est implantée dans SAIDA.

Notre objectif dans ce paragraphe est de montrer comment un enseignant peut expliciter différentes stratégies de raisonnement et mettre éventuellement en évidence des schémas généraux tels les paradigmes étudiés par le groupe Anagram [GRAM,86]. Nous précisons comment implanter une forme de raisonnement à l'aide de la notion d'étape et nous présentons à l'aide de schémas récapitulatifs et d'exemples de dialogues celles qui sont actuellement implantées dans SAIDA.

Implanter un raisonnement à l'aide de la notion d'étape

Nous avons vu dans l'exemple développé au paragraphe précédent comment utiliser la notion d'étape pour regrouper et ordonner les questions à poser à l'utilisateur ; on peut plus largement y regrouper et ordonner les attributs que le système doit valuer au cours de l'étape. Nous abordons maintenant la question de la structuration d'un raisonnement ainsi découpé en étapes et de l'enchaînement des étapes entre elles.

Nous développons un graphe d'étapes dans lequel nous avons toujours un point d'entrée, l'étape de démarrage notée et/départ, et un point de sortie, l'étape terminale notée et/fin ; pour ne pas surcharger les schémas, l'étape terminale n'est jamais dessinée. Ces deux points sont des passages obligés dans la conduite du raisonnement. Il nous faut ensuite décrire un enchaînement d'étapes intermédiaires, nommées sous-étapes. Deux sous-étapes peuvent devoir être exécutées l'une après l'autre, leurs noms se succèdent dans la zone "OBLIGATOIRE" de l'étape qui les englobe. On peut vouloir soumettre l'exécution d'une étape à des conditions liées aux résultats fournis par une autre, de telles étapes figurent dans la zone "ENVISAGEABLE" de la description d'étape ; il faut alors écrire des règles assurant cette gestion des étapes.

Réalisation d'une étape

Exemple : réalisation d'une étape qui a pour objectif de faire acquérir quelques informations sur l'objet, puis d'essayer de conclure successivement sur les différentes représentations possibles.

Etape et/chemin :
OBLIGATOIRE
et/issu/de , et/fichier .
ENVISAGEABLE
et/tab/bool , et/tab/val , et/hachage , et/arbre , et/chaîne .
COND ARRET
arrêt .

Au cours de cette étape, les sous-étapes et/issu/de et et/fichier seront obligatoirement exécutées. Des règles ajoutées à la base de règles permettent de gérer l'enchaînement des

étapes.

Exemple d'une telle règle :

si fin-sortie-et/fichier alors $envisager (et/tab/bool) .

Cette règle exprime le fait que si l'attribut fin-sortie-et/fichier a reçu la valeur vrai à la fin de l'exécution de la sous-étape et/fichier, la sous-étape et/tab/bool sera exécutée. La gestion des autres étapes nommées sur la même ligne est assurée dynamiquement de la même façon, l'ordre d'écriture ne présume en rien de l'ordre d'utilisation. En résumé, on associe à chaque étape des attributs particuliers comme son nom, la fin d'étape, et on écrit avec ces attributs des règles de même nature que celles qui figurent déjà dans la base; on utilise seulement une syntaxe particulière qui permet au moteur d'inférences de repérer ces attributs.

Implantation de raisonnements

La démarche d'étude systématique des contraintes, objets et procédures décrite au paragraphe précédent peut se traduire alors par la structuration d'étapes suivante, rendue apparente à l'utilisateur du système par l'affichage systématique des nom et objectifs de l'étape :

ETAPE et/tout :
ENTREE
! étape et/tout
! on étudie successivement les contraintes, les objets, les procédures
OBLIGATOIRE
et/contraintes , et/objets , et/procédures .
BUTS
conseil1 ,
tableau de booléens indexé par le type de l'élément de l'ensemble ,
tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément ,
tableau trié de valeurs des éléments ,
tableau ordre quelconque de valeurs des éléments ,
liste chaînée avec hachage ,
arbre binaire de recherche ,
liste chaînée de valeurs triée ,
liste chaînée de valeurs dans un ordre quelconque ,
fichier à accès direct .
COND-ARRET
arrêt .
Suivent ici les descriptions des trois sous-étapes utilisées
FIN .

Une démarche permettant de conclure rapidement sur tableaux de booléens ou tableaux de valeurs serait :

ETAPE et/vite :
ENTREE
! étape et/vite
! on essaie de conclure rapidement sur tableaux de booléens ou de valeurs si c'est possible
.
BUTS
tableau de booléens indexé par le type de l'élément de l'ensemble ,
tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément ,
tableau trié de valeurs des éléments , ...
COND-ARRET
arrêt .

La base de connaissances contient des règles provoquant l'arrêt de cette étape dès que l'un des attributs d'implantation proposés comme buts prend la valeur "souhaité" :

si tableau de booléens indexé par le type des éléments de l'ensemble = souhaité alors arrêt .

La gestion des attributs liés aux étapes est assurée comme celle des autres attributs dans la base de faits, et on peut obtenir la valeur de ces attributs.

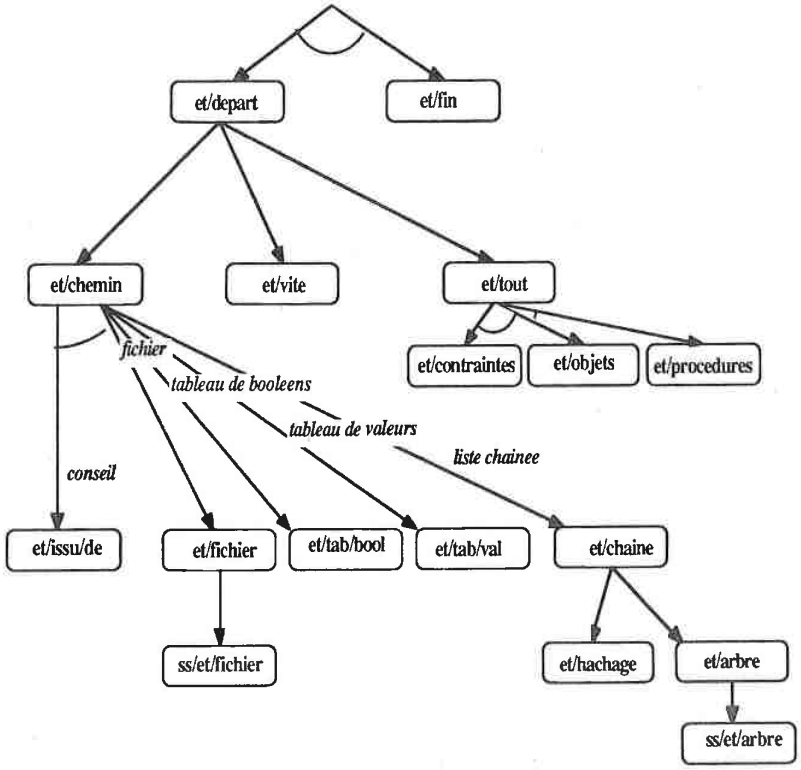
Les raisonnements implantés dans SAIDA

Nous avons implanté dans SAIDA à titre expérimental les trois démarches que nous avons mentionnées :
- démarche guidée par les données implantée par l'étape et/tout,
- démarche guidée par les représentations implantée par l'étape et/chemin,
- démarche guidée par une volonté de conclusion rapide implantée par l'étape et/vite.

Chacune de ces démarches a nécessité l'écriture de sous-étapes et de règles d'inférences associées. Un dialogue détaillé pour l'implantation de MOTS-CLES avec l'étape et/chemin est donné en fin de chapitre, celui qui utilise l'étape et/tout a été fourni en illustration de l'idée de structuration d'univers, la construction de l'étape et/vite a été ébauchée.

La structuration de ces raisonnements en étapes et sous-étapes est schématisée par l'arbre de la figure ci-après. Aux noeuds de cet arbre figurent des étapes. Un arc de E vers E' indique que l'étape E' est une sous-étape de E. Un lien entre deux arcs de même origine et d'extrémités E et E' indique que E' est obligatoirement exécutée lorsque E est achevée. On parle souvent dans les arborescences formalisant des raisonnements de noeud ET. Une

absence de lien indique que E est exécutée la première. Selon les résultats du déroulement de E, E' sera ou ne sera pas exécutée ; on peut parler de noeud OU.



Structuration des raisonnements disponibles dans SAIDA sur l'implantation d'un objet de type ensemble.

3.2.3 D'autres caractéristiques du dialogue

Nous avons dit que nous n'avions pas de modèle explicite de l'élève. Nous proposons cependant des niveaux et des formes de dialogue adaptés à des publics différents, et en l'absence de détermination par le système d'un profil d'élève, nous laissons à l'enseignant la responsabilité de paramétrer le système pour l'adapter aux besoins des utilisateurs élèves. Il est en effet possible de poser des questions plus ou moins détaillées, en utilisant la propriété des attributs dans les systèmes à règles de production d'être demandables ou non au niveau du dialogue.

Exemple : Dans une stratégie de raisonnement dirigée par les représentations, on a besoin de

savoir si le type des éléments de l'ensemble peut servir d'index dans un tableau ADA avant de conclure à une représentation en "tableau de booléens". Pour certains élèves, ce niveau de question peut être jugé suffisant, pour d'autres il faudrait demander de façon exhaustive si le type est entier ou caractères ou énuméré.

Réalisation : dans le premier cas, l'attribut "type des éléments de l'ensemble peut servir d'index en ADA" sera demandable, dans le second cas, il ne le sera pas.

Dans les traces d'exécution présentées, on note la possibilité de répondre "je ne sais pas" à la question posée. La nature des réponses autorisées est un autre facteur sur lequel on peut jouer pour adapter le dialogue à des catégories différentes d'apprenants.

Etudions le cas des réponses "je ne sais pas" et interrogeons-nous sur l'opportunité qu'il y à autoriser ce genre de réponse pour des élèves et à la façon dont un système peut y donner suite. Pourquoi répond-on "je ne sais pas" dans un dialogue lié au choix d'implantation d'objets ? Nous distinguons au moins trois raisons possibles :

- Je ne sais pas parce que je ne dispose pas de l'information demandée.
Exemple : Y a-t-il - des problèmes de place en mémoire centrale - ? La réponse est tout à fait acceptable, mais le système s'en sert pour déduire par défaut qu'il n'y a pas de problème ; l'élève peut en être prévenu immédiatement.

- Je ne sais pas parce que les réponses qu'on me propose sont mal adaptées à mon problème.
Exemple : Avons-nous - plus d'adjonctions/suppressions que de consultations - ? Mon algorithme est complexe, j'ai l'impression qu'il y en a à peu près autant de chaque ! Là encore, la réponse est justifiée, mais l'élève doit en connaitre les conséquences, soit une inférence par défaut, soit d'autres questions permettant d'atteindre le but qui avait provoqué la question.

- Je ne sais pas parce que je ne comprends pas la question ou parce que je ne sais pas y répondre... c'est évidemment plus grave. On peut envisager d'interdire la réponse "je ne sais pas" avec certains attributs pour obliger les élèves à fournir des réponses, on peut également rajouter des règles au système qui provoquent l'impression d'explications lorsque cette valeur a été fournie.

En phase de mise au point du système, il est important de garder une trace des réponses "je ne sais pas" ; elles peuvent simplement révèler de mauvais libellés de questions.

Enfin à tout instant l'élève peut demander les valeurs possibles pour un attribut,

obtenir la liste des déductions faites par le système après l'entrée d'une donnée ainsi que les règles appliquées. Lorsque le système est en phase d'évaluation de but, il peut demander quelle est la règle en cours d'évaluation. On trouve des exemples d'utilisation de ces possibilités dans la trace de dialogue donnée en fin de chapitre.

3.2.4 Les choix de l'enseignant

Nous venons de présenter différentes solutions qui permettent d'agir sur la conduite du déroulement du dialogue et sur des modalités de dialogue pour créer des environnements différents à partir de la base de connaissances construite sur les représentations d'objets abstraits. Nous voulons résumer dans ce paragraphe les choix qui sont ainsi offerts à un enseignant pour adapter un environnement pédagogique général à des objectifs particuliers.

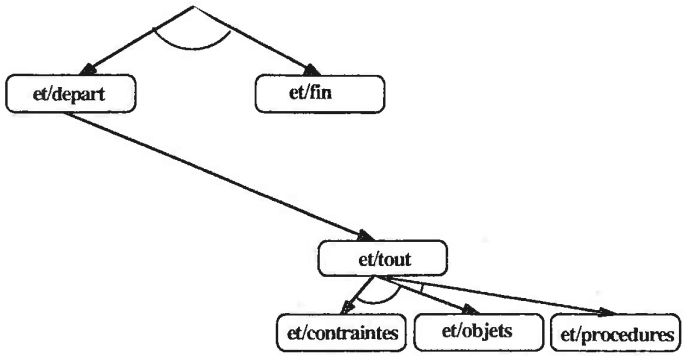
Ce qui suit concerne une base de connaissances "figée" en ce qui concerne les types, implantations de types et critères de choix ; nous avons déjà évoqué au chapitre précédent la question de l'extension d'une telle base et nous considérons ici que cela relève de l'expertise du domaine et non d'une adaptation pédagogique de l'environnement.

Choisir un mode de raisonnement

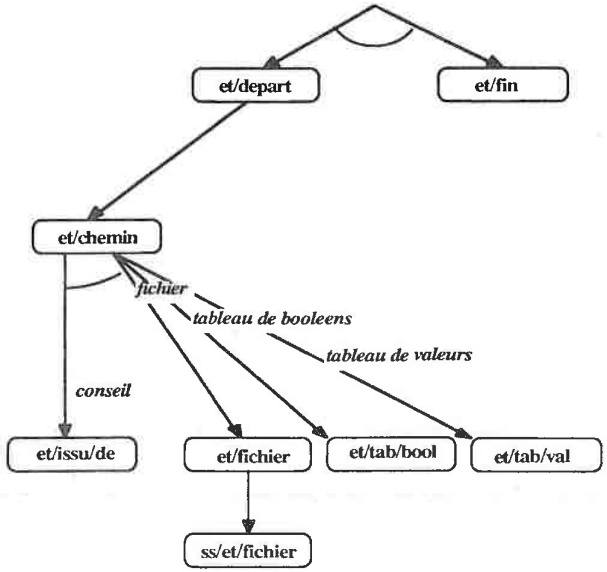
La mise en oeuvre de ces choix différents repose sur des définitions et des enchaînements particuliers d'étapes. Dans l'état actuel de la réalisation, l'enseignant dispose de plusieurs stratégies dans le système et peut en créer d'autres.

S'il utilise les stratégies implantées dans le système, il peut choisir de les mettre toutes à la disposition des élèves. Dans ce cas, il suffit de travailler avec une version de la base de connaissances offrant au démarrage un choix de stratégies de raisonnement. Il peut également n'en choisir qu'une et utiliser une version de la base de connaissances où seule cette stratégie est implantée. Le premier choix était illustré par l'arbre d'étapes donné au paragraphe précédent, le second est illustré par les sous-arbres de la page suivante.

S'il veut créer ses propres stratégies, il doit reconstruire des étapes soit en réordonnant des sous-étapes existantes, soit en créant de nouvelles sous-étapes et écrire les règles de gestion d'étapes correspondantes. Il utilise ainsi les attributs définis pour la base de règles du domaine mais n'a pas à modifier celle-ci. La réalisation technique d'un fichier d'étapes est décrite et illustrée en quatrième partie.



Sous- arbre de raisonnement impliquant une description exhaustive des caractéristiques du problème



Sous-arbre de raisonnement essayant de conclure successivement sur chaque grande famille d'implantations

Choisir les caractères du dialogue

Pour décider du caractère demandable ou non demandable de certains attributs ou de l'autorisation de répondre "je ne sais pas", nous utilisons les fonctionnalités offertes par le générateur de système expert. L'élève, quant à lui, utilise les possibilités qui lui sont offertes d'avoir un dialogue plus ou moins "bavard" ; à ce niveau, nous nous reposons aussi sur les

possibilités du générateur, il faudrait pouvoir décrire ces choix de la même façon que les raisonnements pour que l'enseignant agisse davantage sur le dialogue entre l'élève et le système

Le volet pédagogique de notre système se compose donc :

- de l'utilisation de la base de connaissances construite au chapitre précédent avec notamment:
  * les questions posées
  * les traces de raisonnement fournies
  * la possibilité à tout instant
    - de connaître les valeurs possibles pour l'attribut demandé
    - de demander pourquoi une question est posée

- de conduites de raisonnement conçues à des fins pédagogiques en prenant cette base pour noyau et exprimées sous forme d'étapes et de règles de gestion des étapes

- d'adaptation de niveaux de dialogue entre le système et l'apprenant.

L'expertise pédagogique de ce système est donc, pour l'essentiel, "cachée" dans les étapes et enchaînements d'étapes ajoutés à la base initiale ; elle s'exprimerait évidemment mieux sous une forme plus déclarative, et d'autres requêtes des enseignants pourraient être exprimées, notamment des modifications dans la base de connaissances initiale ; des outils expérimentaux comme AMALIA [VIV,87a] ou SNALI [PAL,88] permettent cette expression .

Nous avons plutôt voulu montrer quel rôle pédagogique on peut faire jouer à l'expertise du domaine, notamment à la structuration de cette expertise, et aux différentes formes de dialogue disponibles avec la plupart des générateurs de systèmes à base de connaissances. Une telle démarche est-elle possible dans d'autres domaines ? Faut-il créer des bases de connaissances spécifiques à l'enseignement ? Nous utilisons cette étude de cas et d'autres travaux pour apporter des éléments de réponses à ces questions en fin de quatrième partie.

Afin d'illustrer ces différentes possibilités, nous donnons au paragraphe suivant une trace détaillée de dialogues issus de l'implantation de l'objet MOTS-CLES.

3.2.5 Exemples de dialogues détaillés

EVALUATION DE L'ETAPE : et/départ
Informations : Réservée à l'enseignant en phase de test du système expert, pour choisir une stratégie de travail. Pour quitter ; entrez "$". Pour sélectionner une étape c'est à dire une méthode : entrez le nom de l'étape que vous souhaitez activer puis entrez "fin". La liste des étapes possibles est :
et/chemin : on suit la constitution de la base
et/vite : on cherche à conclure le plus rapidement possible
et/tout : on examine tous les attributs recensés.
ENTREE: **et/chemin**
ENTREE: **fin**
Contrôle étape : L'étape et/chemin est intéressante en effet c'est vous qui l'affirmez.

EVALUATION DE L'ETAPE : et/chemin
Informations : On suit la démarche qui est celle de la mise en place de la base.
Contrôle étape : L'étape et/issu/de est intéressante car c'est une étape obligatoire pour et/chemin.

EVALUATION DE L'ETAPE : et/issu/de
Informations : On regarde si l'objet étudié est issu d'objets existants. On donne des informations complémentaires.
*Résolution des buts pout l'étape et/issu/de : conseil1 *.*

Question : Avons nous l'ensemble étudié est le résultat d'une recopie ?
Réponse utilisateur : **non**

Question : Avons nous l'ensemble étudié est le résultat d'une opération ensembliste ?
Réponse utilisateur : **non**
*Je déduis : conseil1 = valeur non connue, cet attribut était non demandable. Le but: conseil1 * est vérifié .*
Contrôle étape : L'étape et/issu/de n'est plus intéressante car elle est entièrement évaluée. L'étape et/fichier est intéressante car c'est une étape obligatoire pour et/chemin.

EVALUATION DE L'ETAPE : et/fichier
Informations : Pour savoir si on doit utiliser un fichier.
Question : Avons nous le support est imposé à: fichier accès direct ?
Réponse utilisateur : **non**
Contrôle étape : L'étape ss/et/fichier est intéressante car c'est une étape obligatoire pour et/fichier.

EVALUATION DE L'ETAPE : ss/et/fichier
Question : Avons nous la taille de l'ensemble est trop importante pour être traitée en mémoire centrale ?
Réponse utilisateur : **non**

Je déduis : fichier à accès direct = refusé.
*Contrôle étape : L'étape ss/et/fichier n'est plus intéressante car elle est entièrement évaluée. Je déduis fin-sortie-et-fichier = vrai. Le but: fichier à accès direct * est vérifié, l'étape et/fichier n'est plus intéressante car elle est entièrement évaluée. L'étape et/tab/bool est intéressante, en effet, elle est envisageable pour et/chemin et elle est envisagée (règle r87).*

EVALUATION DE L'ETAPE : et/tab/bool
Informations : Pour conclure sur tableau de booléens
Question : Quelle est la valeur du type de l'élément ?
Réponse de l'utilisateur : **chaîne de caractères**
Je déduis : éléments homogènes = vrai, élément peut servir d'index dans un tableau en ada = faux.

Question : Avons nous les éléments du référentiel sont connus pour chaque exécution ?
Réponse de l'utilisateur : **oui**
Résolution des buts pour et/tab/bool : le but est vers-booléens = vrai.

Question : Avons nous le référentiel de l'ensemble est partagé avec d'autres ensembles ?
Réponse de l'utilisateur : **non**

Question : Avons nous la taille ensemble est en permanance bien plus petite que celle du référentiel ?
Réponse de l'utilisateur : **non**

Question : Avons nous la taille de l'ensemble est voisine de celle du référentiel ?
Réponse de l'utilisateur : **oui**

Je déduis : vers-booléens = faux , il existe une borne connue de la taille de l'ensemble = vrai, tableau de booléens = refusé, conseil2 = valeur non connue, tableau de booléens indexé par le type de l'élément de l'ensemble = refusé, tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément = refusé. Vers-booléens a pour valeur faux, ceci a été démontré par la règle 7: si l'élément peut servir d'index dans un tableau en ada = faux et les éléments du référentiel sont connus pour chaque execution et le référentiel de l'ensemble est partagé avec d'autres ensembles = faux et la taille de l'ensemble est voisine de celle du référentiel alors vers-booleens = faux

*Contrôle étape : Le but vers-booléens * est atteint et l'étape et/tab/bool est verifiée. Je conclus fin-sortie-et/tab/bool = vrai. L'étape et/tab/bool n'est plus intéressante car elle est entièrement évaluée. L'étape et/tab/val est intéressante, en effet, elle est envisageable pour et/chemin et elle est envisagée (règle r89).*

EVALUATION DE L'ETAPE : et/tab/val
Informations : Pour conclure sur un tableau de valeurs.
Les attributs suivants ont déjà une valeur : le type de l'élément = chaîne de caractères, éléments homogènes = vrai, il existe une borne connue de la taille de l'ensemble = vrai
*Le but de l'étape et/tab/val est: tableau de valeurs des éléments *.*

Question : Avons nous beaucoup d'adjonctions/suppressions en position quelconque ?
Réponse de l'utilisateur : **non**

Question : Avons nous il y a beaucoup d'adjonctions ?
Réponse de l'utilisateur : **non**

Question : Quelle est la valeur des variations du nombre d'éléments ?
Réponse de l'utilisateur : **fixe**
Je déduis : tableau de valeurs des éléments = souhaité, vers-chaînée = valeur non connue, arbre binaire de recherche = refusé. Tableau de valeurs des éléments a pour valeur souhaité. Ceci a été demontre grâce à la règle : si éléments homogènes = vrai, et il existe une borne connue de la taille de l'ensemble = vrai, et les variations du nombre d'éléments = fixe alors tableau de valeurs des éléments = souhaité.
*Le but: tableau de valeurs des éléments * est vérifié. Le but à vérifier est: organisation **

Question : Avons nous il existe un ordre sur les éléments ?
Réponse de l'utilisateur : **oui**

Question : Quelle est la valeur de opérations de parcours ?
Réponse de l'utilisateur : **néant**

Question : Avons nous il existe des opérations ensemblistes avec l'objet comme composant?
Réponse de l'utilisateur : **oui**

Question : Avons nous on souhaite favoriser les test d'app/min/max ?
Réponse de l'utilisateur : **oui**
Je déduis : organisation = triée, tableau trié de valeurs des éléments = souhaité, tableau ordre quelconque de valeurs des éléments = valeur non connue, liste chaînée de valeurs dans un ordre quelconque = valeur non connue, arret = vrai, et/chaine = faux, et/chemin = faux.

Contrôle étape : L'étape et/tab/val n'est plus intéressante car elle a été activée par l'étape et/chemin qui a été abandonnée.
L'étape et/chemin n'est plus intéressante car sa condition d'arrêt est vérifiée.
L'étape et/fin est intéressante car c'est une étape obligatoire pour et/départ.

EVALUATION DE L'ETAPE : et/fin
Contrôle étape : L'étape et/fin n'est plus intéressante car elle est entièrement évaluée.

Des informations : Voici les conclusions :
- tableau de booléens indexé par le type de l'élément de l'ensemble = refusé
- tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément= refusé
- tableau trié de valeurs des éléments = souhaité
- arbre binaire de recherche = refusé
- fichier à accès direct = refusé.

Des explications : Pour demander des explications sur un attribut donnez son nom. Pour terminer tapez fin. Pour quitter tapez $.

SORTIE : **fin** .

Etude de la base de faits

Au début de l'étape et/tab/bool :
Variables de la base données (+) ou déduites (-)
+ l'ensemble étudié est le résultat d'une recopie = faux
+ l'ensemble étudié est le résultat d'une opération ensembliste = faux
- conseil1 = valeur non connue car règle de déduction non applicable
+ le support est imposé à fichier accès direct = faux
+ la taille de l'ensemble est trop importante pour être traitée en mémoire centrale = faux
- fichier à accès direct = refusé
Variables de structuration du raisonnement
- et/départ = vrai car l'étape est toujours active
- et/chemin = vrai car l'étape est toujours active
- et/issu/de = faux car terminée
- et/fichier = faux car étape terminée
- ss/et/fichier = faux car sous étape terminée
- fin-sortie-et/fichier = vrai car étape et/fichier terminée
- et/tab/bool = vrai car c'est l'étape courante.

A la fin de l'étape et/tab/bool :
Variables de la base données ou déduites
+ l'ensemble étudié est le résultat d'une recopie = faux
+ l'ensemble étudié est le résultat d'une opération ensembliste = faux
- conseil1 = valeur non connue
+ le support est imposé à fichier accès direct = faux
+ la taille de l'ensemble est trop importante pour être traitée en mémoire centrale = faux
- fichier à accès direct = refusé
+ le type de l'élément = chaîne de caractères
- éléments homogènes = vrai
- élément peut servir d'index dans un tableau en ada = faux
+ les éléments du référentiel sont connus pour chaque exécution = vrai
+ le référentiel de l'ensemble est partagé avec d'autres ensembles = faux
+ taille ensemble est en permanance bien plus petite que celle du référentiel = faux
+ la taille de l'ensemble est voisine de celle du référentiel = vrai
- vers-booléens = faux
- il existe une borne connue de la taile de l'ensemble = vrai
- tableau de booléens = refusé
- conseil2 = valeur non connue
- tableau de booléens indexé par le type de l'élément de l'ensemble = refusé
- tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément= refusé
Variables de structuration du raisonnement
- et/départ = vrai
- et/chemin = vrai
- et/issu/de = faux
- et/fichier = faux
- ss/et/fichier = faux
- fin-sortie-et/fichier = vrai
- et/tab/bool = faux
- fin-sortie-et/tab/bool = vrai
- et/tab/val = vrai.

Des informations en fin d'interrogation

Voici les conclusions :
- conseil1 = valeur non connue
- tableau de booléens indexé par le type de l'élément de l'ensemble = refusé
- tableau de booléens indexé par un intervalle d'entiers avec recherche préalable du rang de l'élément= refusé
- conseil2 = valeur non connue
- **tableau trié de valeurs des éléments = souhaité**
- tableau ordre quelconque de valeurs des éléments = valeur non connue
- pas de valeur pour l'attribut: liste chainée avec hachage
- arbre binaire de recherche = refusé
- pas de valeur pour l'attribut: liste chainée de valeurs triée
- liste chainée de valeurs dans un ordre quelconque = valeur non connue
- fichier à accès direct = refusé.

Exemple de dialogue entre utilisateur et système expert

Question : Quelle est la valeur de opérations de parcours ?
Réponse de l'utilisateur : **valeurs possibles**
Réponse du système : néant, exclusivement, majoritairement
Réponse de l'utilisateur : **néant**
Je déduis : ...

Question : Avons nous il existe des opérations ensemblistes avec l'objet comme composant ?
Réponse de l'utilisateur : **pourquoi**
Réponse du système : Je cherche organisation
Je sais déjà que : il existe un ordre sur les éléments = vrai si de plus il existe des opérations ensemblistes avec l'objet comme composant = vrai alors organisation = triée
Réponse de l'utilisateur : **oui**
Je déduis : organisation = triée , ...

Un exemple d'explications demandées en fin d'interrogation

Des explications :
Pour demander des explications sur un attribut donnez son nom.
Pour terminer tapez fin. Pour quitter tapez $.
SORTIE : **tableau trié de valeurs des éléments**
- - tableau trié de valeurs des éléments = souhaité (règle r27)
- - - - tableau de valeurs des éléments = souhaité (règle r10)
- - - - - - éléments homogènes (règle r8)
- - - - - - - le type de l'élément = chaine de caractères (affirmation)
- - - - - - il existe une borne connue de la taille de l'ensemble (règle 81)
- - - - - - - la taille de l'ensemble est voisine de celle du référentiel (affirmation)
- - - - - les variations du nombre d'éléments = fixe (affirmation)
- - - - organisation = triée (règle r16)
- - - - - il existe un ordre sur les éléments (affirmation)
- - - - - on souhaite favoriser les test d'app/min/max (affirmation).

SORTIE : **tableau de booléens indexé par le type de l'élément de l'ensemble**
- tableau de booléens indexé par le type de l'élément de l'ensemble = refusé (règle r85)
- - tableau de booléens = refusé (règle r84)
- - - non vers-booléens (règle r4)
- - - - non élément peut servir d'index dans un tableau en ada (règle r87)
- - - - - le type de l'élément = chaîne de caractères (affirmation)
- - - - les éléments du référentiel sont connus pour chaque exécution (affirmation)
- - - - non le référentiel de l'ensemble est partagé avec d'autres ensembles (affirmation)
- - - - la taille de l'ensemble est voisine de celle du référentiel (affirmation)

SORTIE : **arbre binaire de recherche**
- arbre binaire de recherche = refusé (règle r35)
- - les variations du nombre d'éléments = fixe (affirmation)

SORTIE : **fichier à accès direct**
- fichier à accès direct = refusé (règle 49)
- - non le support est imposé à fichier à accès direct (affirmation)
- - non la taille de l'ensemble est trop importante pour être traitée en mémoire centrale (affirmation)

SORTIE : **fin** .

L'ENVIRONNEMENT D'ENSEIGNEMENT SAIDA FONCTIONNALITES ET REALISATION

IV° Partie

Introduction à la partie IV

Nous avons présenté dans la troisième partie deux des principaux composants du système SAIDA, la bibliothèque de types et la base de connaissances sur les implantations de types sous ses deux aspects, expertise du domaine et expertise pédagogique.

Cette quatrième partie est consacrée à la description de l'ensemble de l'environnement SAIDA. Nous le présentons en adoptant d'abord le point de vue des utilisateurs, puis celui du réalisateur.

Dans la plupart des logiciels pédagogiques, ou bien l'élève se trouve face à un système rigide ne proposant aucune adaptation à l'apprenant, ou bien il se voit offrir des menus entre lesquels il choisit lui-même, ou enfin, plus rarement, le comportement du système s'adapte aux besoins de l'apprenant à l'aide d'un modèle plus ou moins élaboré de ce dernier. Nous pensons, notamment pour le domaine qui nous concerne, que nous ne disposons pas d'études et de modèles permettant une adaptation très poussée des formes de raisonnement et de dialogue différentes selon les apprenants. L'intervention de l'enseignant nous paraît être dans ce cas une bonne solution pour paramètrer un système et l'adapter d'une part à la progression pédagogique qu'il a choisie, d'autre part au profil d'une population d'élèves.

SAIDA a donc deux catégories bien distinctes d'utilisateurs, des enseignants et des élèves, auxquels il offre un certain nombre de fonctionnalités. La présentation de ces fonctionnalités et leur illustration par des exemples de déroulement de sessions font l'objet du premier chapitre.

Le second chapitre est consacré à l'architecture interne du système. Nous y expliquons notre choix d'utiliser chaque fois que possible des logiciels existants, tout en notant les difficultés d'intégration que l'on rencontre dans une telle entreprise ; nous décrivons ensuite les principales caractéristiques des outils retenus.

Nous avons indiqué à plusieurs reprises que nous envisagions la construction de ce système à la fois pour répondre à un besoin dans un domaine où il n'existait pas de logiciel pédagogique, mais aussi comme étude de cas permettant de faire émerger des idées qui dépassent le cadre de notre application. Après la description de SAIDA, un troisième chapitre pose donc de façon plus générale la question de la construction d'environnements d'enseignement autour de systèmes à bases de connaissances ; il apporte à ce sujet un cadre de réflexion pour la construction de nouvelles applications et souligne des problèmes conceptuels ou techniques qui devront être résolus si l'on veut pouvoir créer facilement des applications pédagogiques dans beaucoup de disciplines et pour des catégories variées d'apprenants.

Chapitre 1

LES FONCTIONNALITES OFFERTES AUX ELEVES ET AUX ENSEIGNANTS

SAIDA s'adresse à des enseignants qui doivent configurer cet environnement pour des élèves. Nous commençons donc par préciser à quels élèves cette application est destinée et pour quels apprentissages ; puis, nous présentons successivement les fonctionnalités offertes aux enseignants et leur réalisation dans le système, et les fonctionnalités offertes aux élèves et leur réalisation dans le système. L'ensemble est illustré par des copies d'écrans commentées issues de sessions d'utilisation.

1.1 LES OBJECTIFS DU SYSTEME

1.1.1 Le public visé

En formation initiale, le public visé par cet environnement est celui des étudiants en informatique ayant des connaissances de base en construction de programmes et la pratique d'un langage comme PASCAL. Notre système peut alors être le support de travaux dirigés de modules de programmation avancée ; les contenus de tels modules d'enseignement comprennent l'entraînement à la programmation abstraite, les types abstraits de données et leurs représentations, l'étude d'algorithmes classiques, la complexité des algorithmes, etc... comme nous l'avons étudié dans la seconde partie de cet ouvrage.

En formation continue, de nombreux programmeurs ont une grande pratique de l'écriture de programmes avec les structures de données de leur langage habituel (COBOL, FORTRAN, BASIC). Nous pensons que s'ils sont amenés à travailler en ADA, le danger est grand de les voir utiliser le sous-ensemble PASCAL, voire FORTRAN, de ce langage. Toute formation continue à ADA doit donc nécessairement comprendre un entraînement à l'écriture d'algorithmes au niveau d'abstraction que permettent les concepts de paquetage et de généricité mis en oeuvre. SAIDA peut constituer un support pour de telles formations.

1.1.2 Objectifs pédagogiques

SAIDA est un environnement de travail ouvert ; il a été conçu dans un esprit de compromis entre les environnements d'assistance à la programmation de demain tels qu'on peut les imaginer à partir des chapitres 2 et 3 de la première partie, et ce que l'on peut réaliser aujourd'hui pour un contexte d'apprentissage n'ayant pas les contraintes de la production.

Il n'est lié ni à une méthode de programmation particulière, ni à un domaine spécifique d'application ; il permet de construire des programmes, mais aussi d'en lire et d'en réutiliser. Il utilise ADA et favorise donc l'apprentissage de la plupart des concepts avancés de programmation implantés dans ce langage. Parmi des modes d'utilisation possibles dans un enseignement d'informatique, nous suggérons notamment les suivants :

- Ecriture d'algorithmes abstraits

L'accent est mis uniquement sur la mise en point d'algorithmes utilisant les types de la bibliothèque SAIDA avec une implantation standard. On ne s'intéresse pas aux questions d'implantation des objets abstraits.

- Comparaison d'implantations

Dans ce deuxième mode, l'élève a à sa disposition plusieurs implantations pour les types qu'il étudie ; l'objectif est de lui permettre de comparer rapidement des implantations différentes, d'effectuer dans son algorithme des mesures, d'expérimenter les notions de complexité liées aux choix d'implantation physique des objets d'un algorithme.

- Choix raisonné d'une implantation

Ce mode de travail utilise le système à base de connaissances de SAIDA et a pour objectif la mise en évidence des critères à prendre en compte pour choisir une implantation efficace dans un contexte donné.

- D'autres approches

Le système peut aussi être utilisé comme environnement de programmation à compléter, notamment par des types propres à un domaine d'application donné ou comme modèle de bibliothèque de paquetages et de textes ADA (style, utilisation de concepts avancés).

C'est pour ces publics potentiels et avec ces objectifs pédagogiques qu'on été définies les fonctionnalités que nous décrivons dans les deux paragraphes suivants. SAIDA comporte un mode élève et un mode enseignant. Le choix entre ces deux modes est effectué grâce à la sélection du bouton approprié sur le premier écran qui s'affiche lorsqu'on lance l'application.

1.2 LES FONCTIONNALITES OFFERTES AUX ENSEIGNANTS

Nous avons indiqué à plusieurs reprises l'importance que nous attachions aux possibilités données à l'enseignant pour déterminer à partir d'un ensemble plus vaste la configuration du système qu'il veut mettre à la disposition de ses élèves. Nous détaillons dans ce paragraphe les fonctionnalités effectivement offertes à l'enseignant dans la version actuelle de SAIDA et concluons en évoquant celles qu'on souhaiterait y ajouter.

1.2.1 Les besoins de l'enseignant

SAIDA est un environnement logiciel dont les ressources sont partagées par plusieurs utilisateurs. Pour définir plusieurs configurations particulières adaptées aux profils de diverses catégories d'élèves, l'enseignant doit donc disposer :

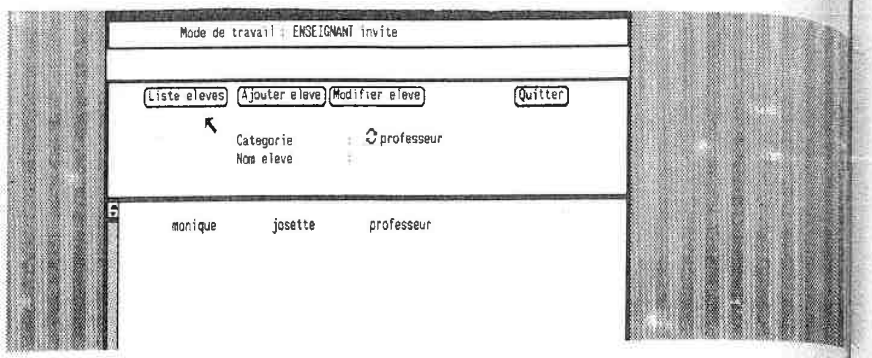
- de fonctions de répartition d'élèves en catégories et de définition et de gestion de caractères propres à chaque catégorie (droits d'accès, archivage de traces de travail, etc...),
- de fonctions permettant de définir le sous-ensemble de la matière à enseigner sur lequel une catégorie d'élèves peut travailler,
- de fonctions offrant, pour un contenu donné, la possibilité de choisir entre des modes d'approche différents,
- de fonctions permettant de faire varier les contenus disciplinaires et les méthodes proposées pour travailler sur ces contenus.

SAIDA offre à l'enseignant l'implantation de fonctions appartenant aux trois premières catégories citées.

1.2.2 Fonctionnalités de "gestion pédagogique des élèves "

Dans SAIDA, l'enseignant peut définir des catégories d'élèves utilisateurs. Pour chaque catégorie, il donne la liste des noms d'élèves et des caractères attachés à cette catégorie. Actuellement ne sont implantés à ce niveau que les droits d'accès des élèves de la catégorie vis à vis des bibliothèques de spécification et d'implantations de types en ADA et le ou les modes de raisonnements à choisir parmi ceux qui sont disponibles dans la base de connaissances.

Exemple de réalisation : l'écran de la figure suivante montre les boutons permettant d'afficher la liste des élèves d'une catégorie donnée et de la modifier. Le cas où un élève figure dans plusieurs listes est prévu : il se voit affecter les caractéristiques de la catégorie de numéro le plus bas dans l'ordre de définition des catégories.



1.2.3 Fonctionnalités permettant de choisir un contenu

- Choix des types

Nous avons vu dans la seconde partie que l'ordre dans lequel les types étaient introduits dans les manuels n'était pas toujours le même. Nous donnons donc à l'enseignant la possibilité de définir la liste des types qu'il veut rendre "visibles" à ses élèves, les autres ne seront pas accessibles. Concrètement, il s'agit de ne donner accès qu'à une partie de la bibliothèque et une partie de la base de connaissances.

- Choix des implantations pour un type

De même, toutes les implantations proposées dans SAIDA peuvent ne pas avoir été vues en cours et l'enseignant peut souhaiter que les élèves ne dispersent pas leur attention sur des notions non étudiées. Pour chaque type de données, le professeur peut donc choisir les implantations qui seront visibles et utilisables par chaque catégorie d'élèves. Là encore il s'agit de ne donner accès qu'à une partie des bibliothèques. Par contre, ce choix n'est pas actuellement répercuté sur la base de connaissances ; sa prise en compte nécessite en effet une reconfiguration profonde des règles associées au type considéré et n'est envisageable que si l'on dispose d'outils automatiques ou semi-automatiques pour le faire.

1.2.4 Fonctionnalités permettant de choisir un mode de travail

Il nous a semblé intéressant de paramétrer non seulement les contenus mais aussi d'autres facteurs qui permettent d'utiliser le logiciel pour des types d'exercices différents et pour des élèves auxquels on veut faire pratiquer des modes de raisonnements différents.

Une forme de travail peut être de faire écrire les modules ADA implantant les

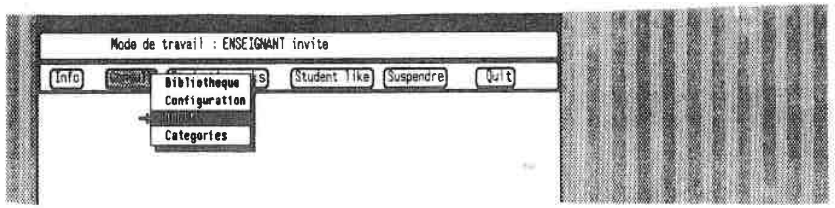
opérateurs attachés à un type pour une ou plusieurs représentations. Le système donne la possibilité de rendre certaines implantations invisibles et inaccessibles pour que ce type d'exercice soit possible.

Une autre forme de travail peut être uniquement l'entraînement à l'écriture d'algorithmes abstraits, et au test rapide des programmes en résultant, sans attention portée aux algorithmes implantant les opérations. Dans ce cas, on peut laisser visible la partie spécification de la bibliothèque et rendre non lisible mais utilisable la partie implantation.

Enfin, dans la base de connaissances, plusieurs modes de raisonnement sont prévus (dirigé par les données, par les résultats, avec plus ou moins d'intermédiaires). Ils peuvent être tous disponibles et laissés au choix de l'élève, mais l'un d'eux peut être imposé par le professeur dans la configuration du système.

1.1.5 Exemple de réalisation

Sur l'écran suivant, l'enseignant vient de sélectionner la fonction "droits" dans le menu. Cette fonction va lui permettre de définir les droits d'accès aux bibliothèques qu'il donne à chaque catégorie d'élèves.

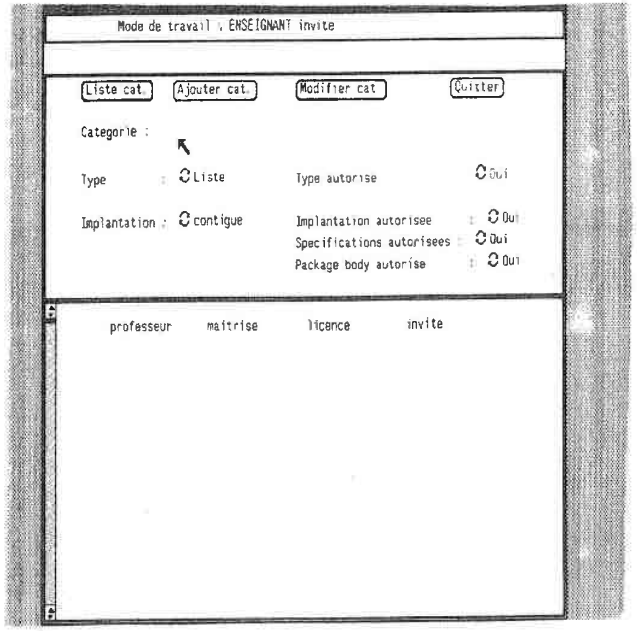


La figure suivante montre l'écran qui permet d'affecter les droits. Pour chaque rubrique type et implantation, le symbole permet de faire défiler la liste des valeurs disponibles ; à droite de l'écran ce même symbole permet de passer de oui à non. L'affectation des droits d'accès pour une catégorie d'élèves à chaque modèle envisagé est ainsi très rapide à réaliser sur un seul écran à l'aide de la souris.

En conclusion, nous avons réalisé l'implantation de fonctions donnant à l'enseignant des possibilités d'adaptation variées et illustrant ainsi l'idée qu'un environnement d'enseignement bien conçu et largement paramétrable peut être utilisé dans de nombreux cadres pédagogiques.

Aux fonctions actuellement disponibles sous forme de menus et boutons, il faut ajouter celles décrites dans les chapitres 2 et 3 de la troisième partie. Elles ne font pas encore l'objet d'implantation par menus parce qu'elles impliquent l'écriture de modules nouveaux de

bibliothèque ADA ou de base de connaissances sous MORSE. L'enseignant dispose d'un système ouvert dans lequel il peut modifier la bibliothèque de types et la base de connaissances. Les indications techniques nécessaires à ces opérations se trouvent dans la notice technique de description du logiciel [MOR, 88].



La modification du contenu des bibliothèques peut se traduire par :

- l'adjonction de nouveaux types,
- l'adjonction de nouvelles implantations pour un type existant,
- le changement des algorithmes implantant certaines opérations,
- l'adjonction ou le retrait d'opérations.

La modification de la base de connaissances peut porter sur l'expertise en critères de choix d'implantations ou sur l'expertise pédagogique. La version actuelle de SAIDA ne fournit aucune aide pour de telles modifications.

Bien d'autres fonctions seraient envisageables : la forme de trace du travail des élèves pourrait être un autre des paramètres du système. On peut garder tous les fichiers créés par les élèves, des copies des dialogues conduits avec le système à base de connaissances ou seulement des renseignements extraits de ces fichiers. Les informations issues de ces traces permettraient un suivi des élèves et l'envoi de messages personnalisés à chacun d'eux.

1.3 LES FONCTIONNALITES OFFERTES AUX ELEVES

Nous voulons rappeler au début de ce paragraphe que nos élèves sont ou seront des informaticiens professionnels et qu'il est donc capital de les laisser travailler avec les outils de production de programme de la profession ; un environnement d'enseignement pour le niveau de cours que nous envisageons, à savoir programmation avancée, ne doit pas inhiber les logiciels disponibles ; il doit plutôt apporter des aides et des guides dont il faudra éventuellement apprendre à se passer et un cadre permettant d'organiser le travail. C'est l'objectif que nous poursuivons au travers des fonctionnalités offertes aux élèves.

1.3.1 L'accès à l'environnement UNIX

L'élève peut à tout instant suspendre le dialogue que le système lui propose, ouvrir une nouvelle fenêtre et travailler dans l'environnement UNIX en disposant de tous les logiciels et toutes les applications disponibles dans cet environnement. Il peut ainsi par exemple :

- consulter des fichiers
- envoyer du courrier électronique
- compléter un texte
- provoquer une impression
- compiler, éditer ou exécuter le programme de son choix
- etc...

Les autres fonctionnalités offertes sont destinées à lui apporter une assistance particulièrement adapté aux travaux pour lesquels SAIDA est prévu.

1.3.2 Création et édition de programmes

A priori, l'étudiant va écrire avec notre système des procédures ADA utilisant les types fournis par la bibliothèque de SAIDA. C'est pour des tâches de cette nature que nous lui fournissons une assistance un peu plus élaborée que celle des environnements UNIX et EMACS.

- Le bouton EDIT permet sans quitter SAIDA d'éditer le fichier de son choix et de travailler dessus sous éditeur.

- Le bouton SEE permet d'ouvrir une autre fenêtre dans laquelle on indique le nom du fichier que l'on veut consulter ; dans cette seconde fenêtre aucune modification n'est permise. Nous pensons là à la possibilité de parcourir des modules de la bibliothèque, par exemple pour

prendre connaissance de la liste d'opérateurs disponibles pour un type abstrait, ou du profil d'un opérateur, afin de poursuivre la création de la procédure figurant dans la fenêtre principale.

1.3.3 Choix d'implantation

SAIDA est doté d'un système à base de connaissances fonctionnant par dialogue avec l'utilisateur, lui proposant une ou des implantations pour un objet abstrait dont il vient de définir les caractéristiques et lui expliquant les raisons ayant conduit à ces propositions.

A l'exception de quelques règles particulières, ce système n'est pas directement lié à ADA ; nous pensons qu'il peut être utilisé :
- pour rechercher des conseils à l'implantation d'objets dans d'autres langages (PASCAL, MODULA,...),
- pour travailler avec la base de connaissances sans avoir immédiatement un programme à compléter.

Le bouton EXPERT permet ce mode d'utilisation

1.3.4 Aides à l'implantation

L'objectif visé est cette fois d'utiliser le système à base de connaissances et de réaliser l'implantation de l'objet étudié en complétant le texte ADA en cours d'écriture. Le système offre alors la possibilité d'interroger le système expert, rappelle les implantations autorisées à l'élève pour le type choisi et aide à réaliser les modifications de texte ADA.

Le bouton IMPLANTATION permet d'accéder à ces différentes fonctions.

Deux boutons sont alors disponibles et se complètent :
CHERCHER OBJET et AIDES A L'IMPLANTATION
Quel est en effet le problème de l'étudiant à ce stade de travail ?

Exemple 1 :
Soit l'objet TEXTE à implanter avec la déclaration suivante dans la procédure ADA :

```
-----------------
TEXTE : LISTE ;   -- texte est une liste de chaînes de caractères
-----------------
```

Cette déclaration suppose une instanciation de paquetage de liste correspondant à l'implantation choisie, par exemple chaînage simple. Le système demande le type des éléments de la liste et produit le texte :


```
PACKAGE LIS_CHAINE_CHN is new LIS_MANAGER_CHN (CHAINE) ; use LIS_CHAINE_CHN ;
```

et le copie à l'emplacement désigné par le curseur de l'utilisateur.

Exemple 2 :
Soit l'objet TEXTE à implanter comme liste chaînée et l'objet PHRASE à implanter comme liste contiguë avec le texte ADA suivant :

```
----------
TEXTE, PHRASE : LISTE ;...
----------
ADJL (TEXTE, ...) ;
----------
```

Il faut instancier deux types liste, un pour chaque objet, et copier les lignes précisant les noms des deux paquetages :

```
PACKAGE LIS_CHAINE_CHN is new LIS_MANAGER_CHN (CHAINE) ; use LIS_CHAINE_CHN ;
```

Le nom de paquetage est proposé par le système ; il est formé à partir du nom du type LIS, du nom du type de l'élément CHAINE et de la représentation choisie CHN. Le système propose donc pour le second paquetage :

```
PACKAGE LIS_CHAINE_CTG is new LIS_MANAGER_CTG (CHAINE) ; use LIS_CHAINE_CTG ;
```

Il faut ensuite repérer, pour chaque objet, les identificateurs liées au type, à commencer par la déclaration LISTE, et les préfixer par LIS_CHAINE_CTG pour ceux qui concernent PHRASE et par LIS_CHAINE_CHN pour ceux qui concernent TEXTE.

Exemple : LIS_CHAINE_CHN. ADJL (TEXTE, ...)

L'usage alternatif des boutons "CHERCHER OBJET" et "COPIER" <nom de paquetage> permet de faciliter cette opération fastidieuse.
Des exemples d'écrans offrant ces boutons sont donnés en figures 10, 11, 12 de la session commentée.

1.3.5 Commandes abrégées

Une fois les choix d'implantation effectués et les procédures ADA complétées, il reste à compiler, lier et exécuter le programme construit.

Pour chacune de ces opérations, l'utilisation du bouton fourni par SAIDA évite de frapper directement sous UNIX des commandes plus longues et accélère le test effectif des programmes.

1.4 UNE SESSION ELEVE COMMENTEE SUR ECRANS

Figure 1.

Une première fenêtre montre l'écran d'accueil ; à ce stade, l'utilisateur donne son nom, et, éventuellement se fait reconnaître comme enseignant. Les écrans qui suivent ne concernent que le mode étudiant.

Sur la deuxième fenêtre apparaît le menu disponible sous forme de boutons, le mode d'emploi s'affiche si l'on active le bouton INFO.

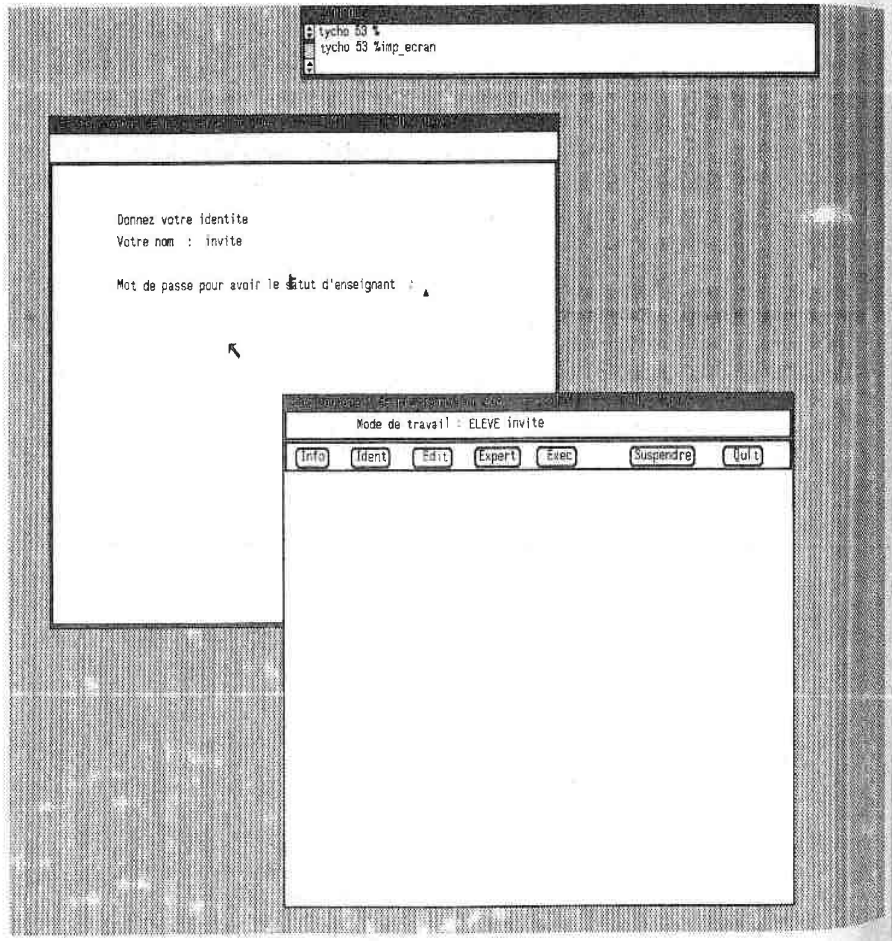


Figure 2.

Ecran obtenu après activation du bouton INFO, on notera la transformation de ce bouton en FINFO pour sortir en l'activant à nouveau.

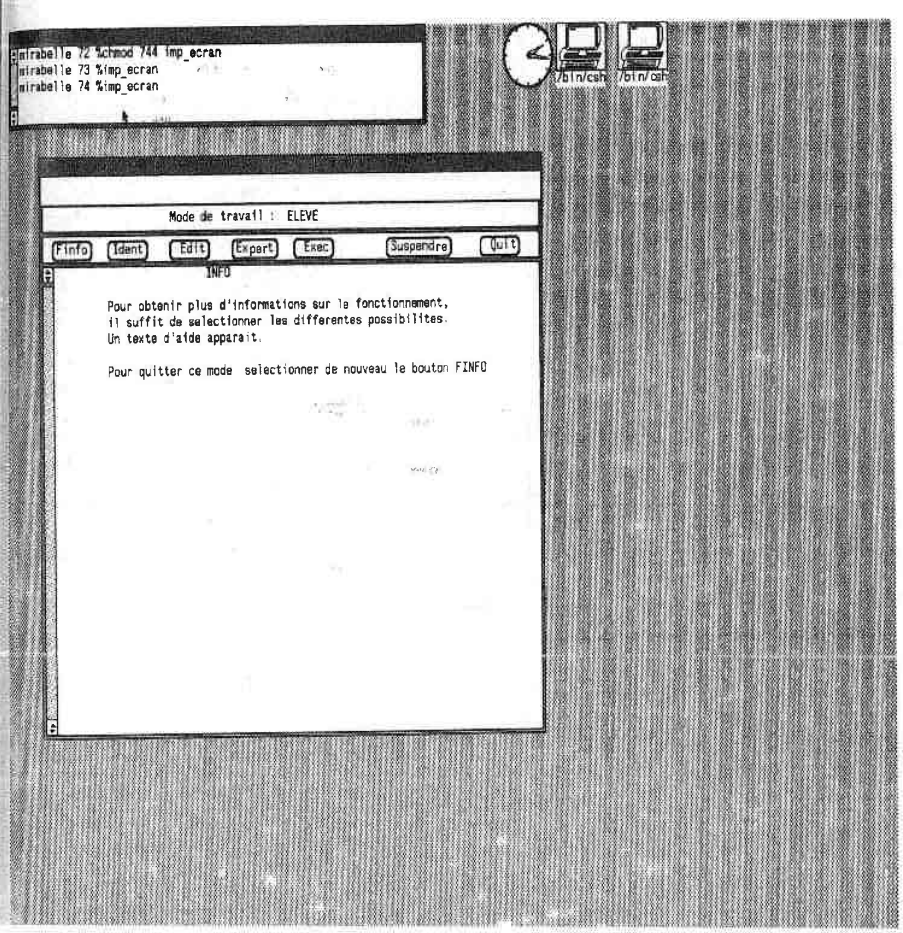


Figure 3.

Ecrans d'informations correspondant aux boutons EXEC et EXPERT.

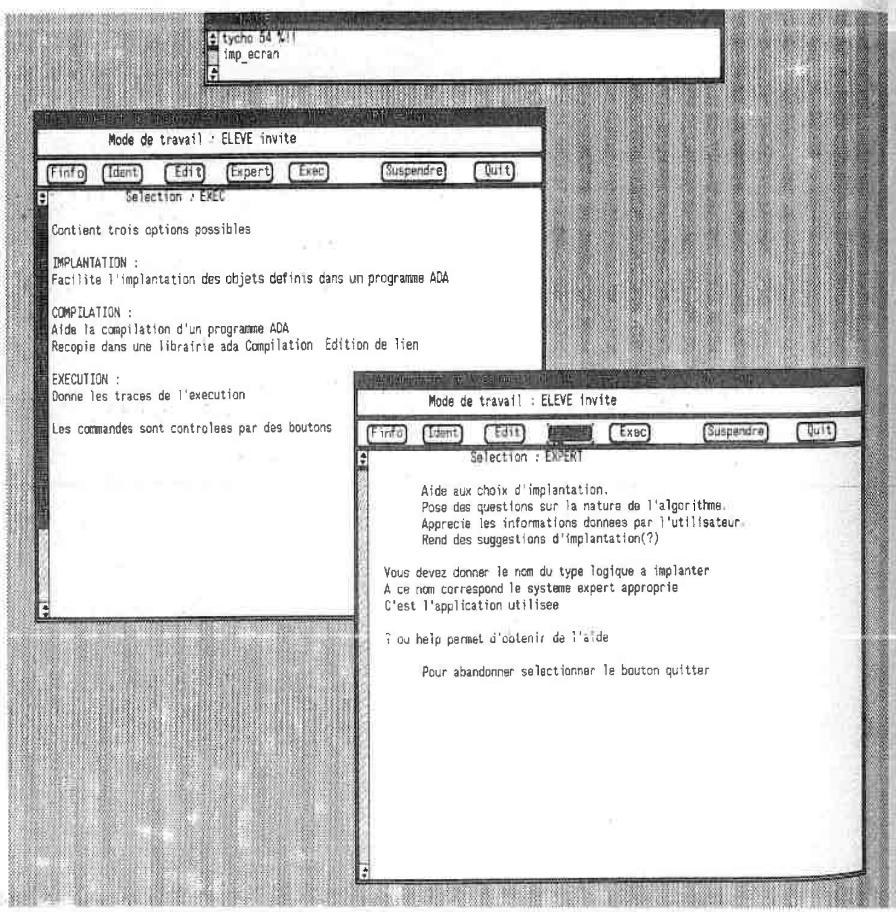


Figure 4.

L'élève obtient cet écran après avoir sélectionné le bouton EDIT pour travailler en mode "édition et création de programme"; dans cet exemple, il a en outre sélectionné FIND puis donné le nom du fichier sur lequel il veut travailler (aprog/compte_mots.a).

A ce niveau, les boutons proposés ont les rôles suivants:

NEW : créer un nouveau fichier,
FIND : retrouver un fichier dont on donne le nom,
SAVE : sauver le fichier courant sous un nom donné,
SEE : consulter sur une autre fenêtre un autre texte.

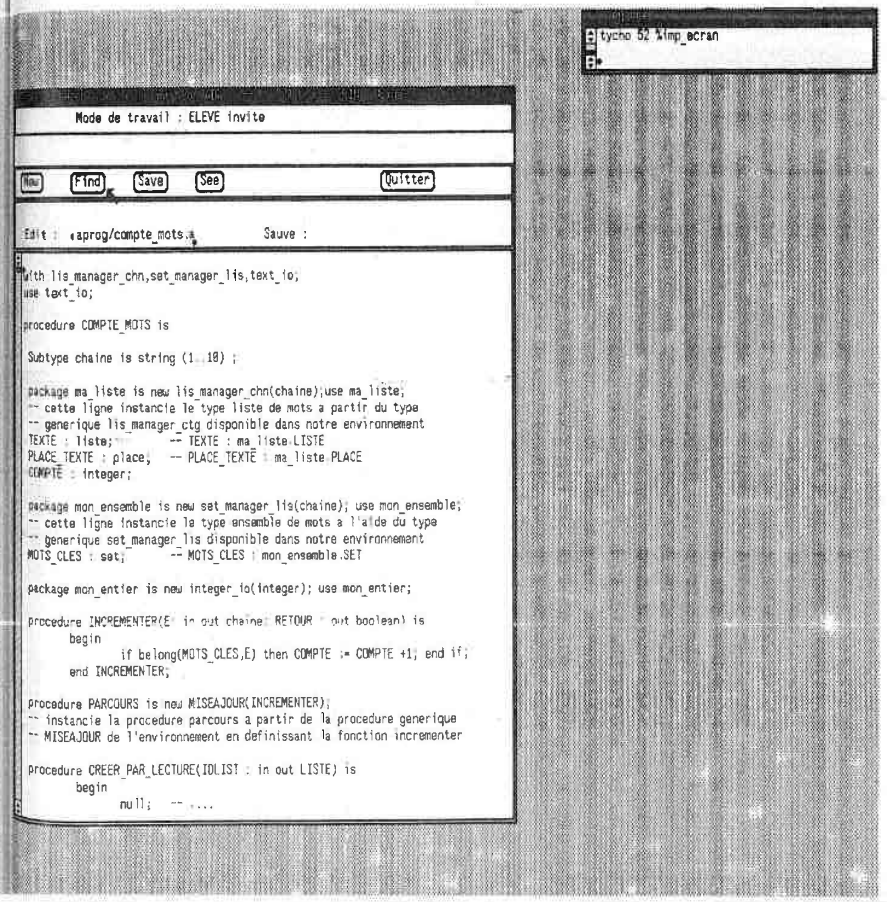


Figure 5.

On note sur la première fenêtre la disparition du bouton SEE qui vient d'être sélectionné et a eu pour effet l'affichage de la deuxième fenêtre; dans cette fenêtre, on a indiqué le nom du texte désiré. L'utilisateur peut se déplacer dans le texte de chacune des fenêtres en faisant "rouler" le curseur des colonnes de gauche. Aucune modification n'est autorisée sur le texte de la deuxième fenêtre.

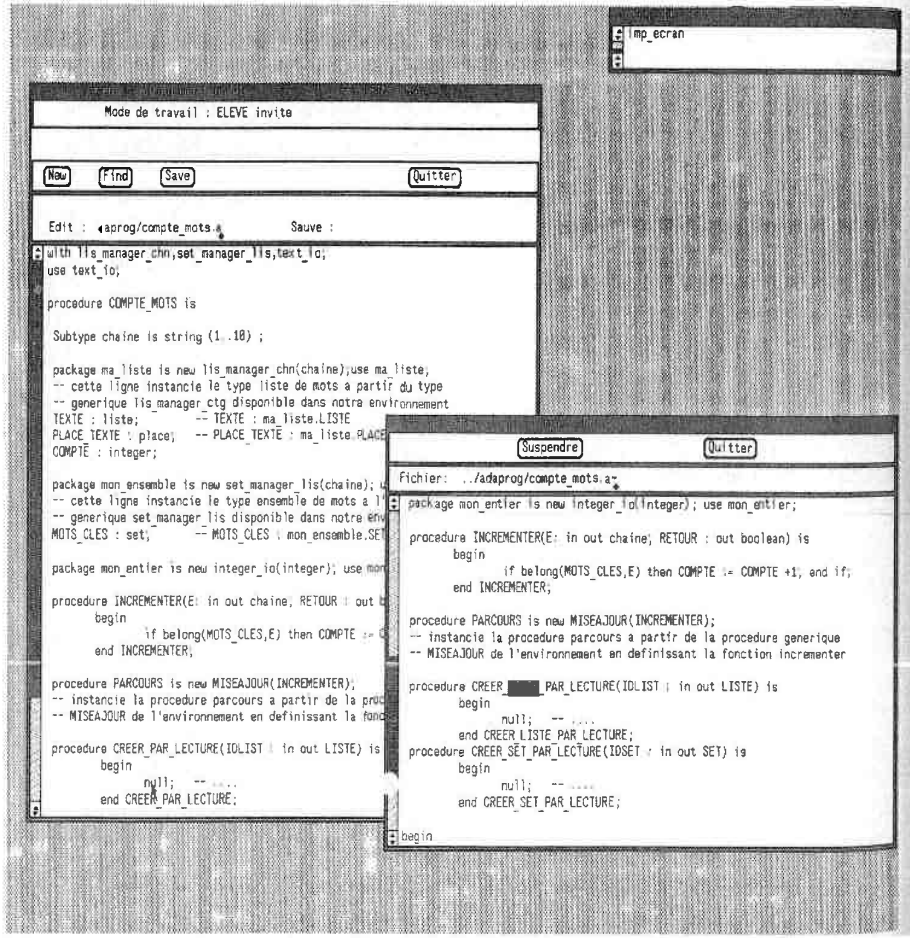


Figure 6.

Cet écran apparait à l'étudiant lorsqu'il choisit d'éxécuter un programme; il peut à ce niveau activer :

IMPLANTATION : pour compléter son texte ADA par les noms de packages à utiliser effectivement (implantations ADA des types abstraits utilisés),

COMPILATION : pour compiler le programme ainsi construit,

EXECUTION : pour éxécuter un programme ADA,

QUITTER : pour revenir au niveau précédent dans l'arborescence des menus.

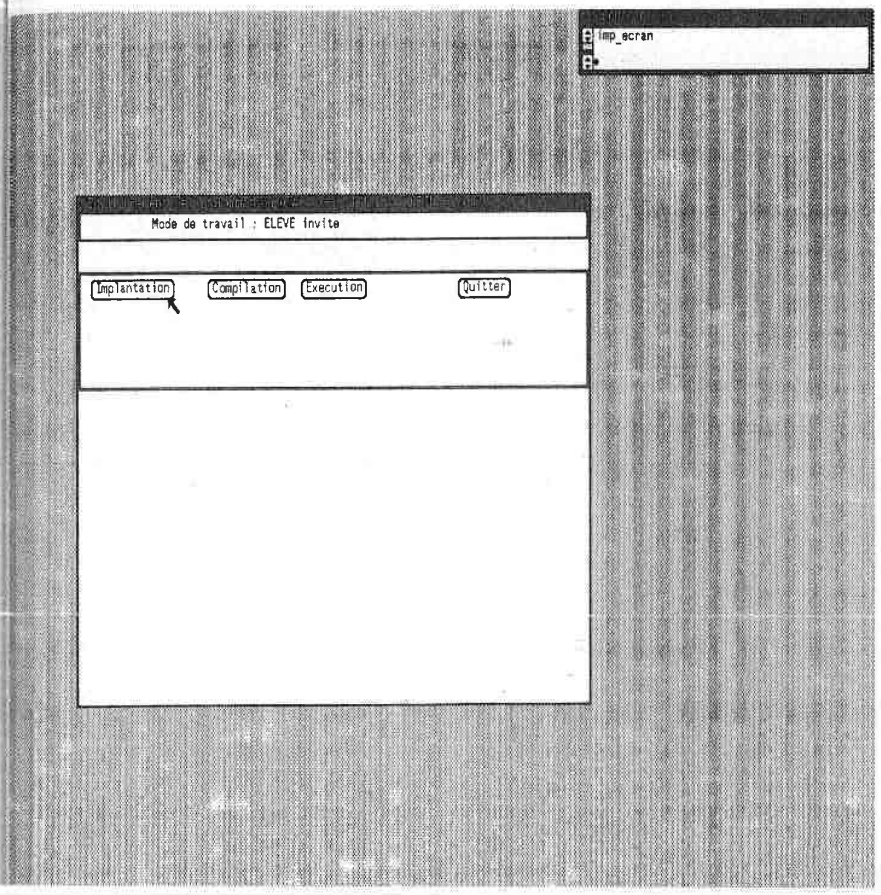


Figure 7.

L'élève a sélectionné IMPLANTATION. A ce niveau, il cherche à déterminer une implantation pour chacun des objets déclarés à l'aide d'un type abstrait dans sa prodédure. Il doit donner l'identification qui désigne l'objet à implanter, ici TEXTE. Il peut ensuite utiliser :
CHERCHER-OBJET : pour trouver plus facilement l'objet dans le texte
AIDES A L'IMPLANTATION pour sélectionner un type et regarder les implantations disponibles.

Enfin, en haut de l'écran, le bouton EXPERT est disponible pour appeler l'application système expert et être guidé dans le choix d'une représentation (cf. figure 10).

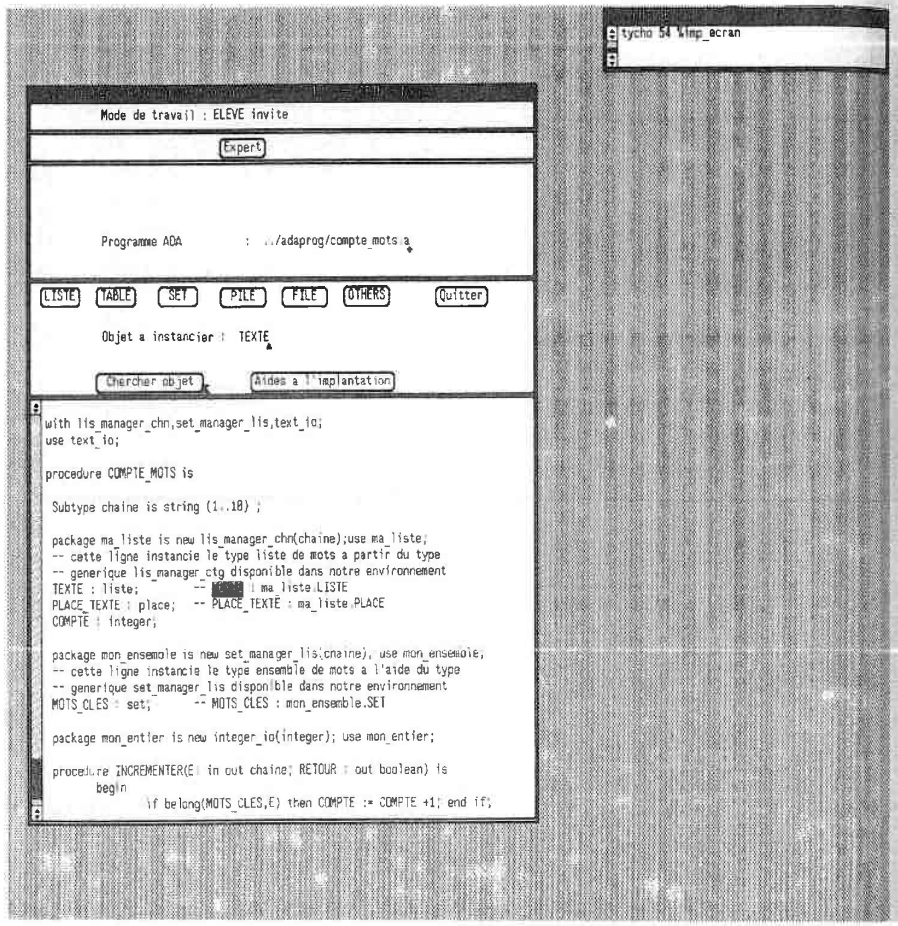


Figure 8.

On a atteint la fin de la procédure COMPTE-MOTS pour la recherche de l'identificateur TEXTE.

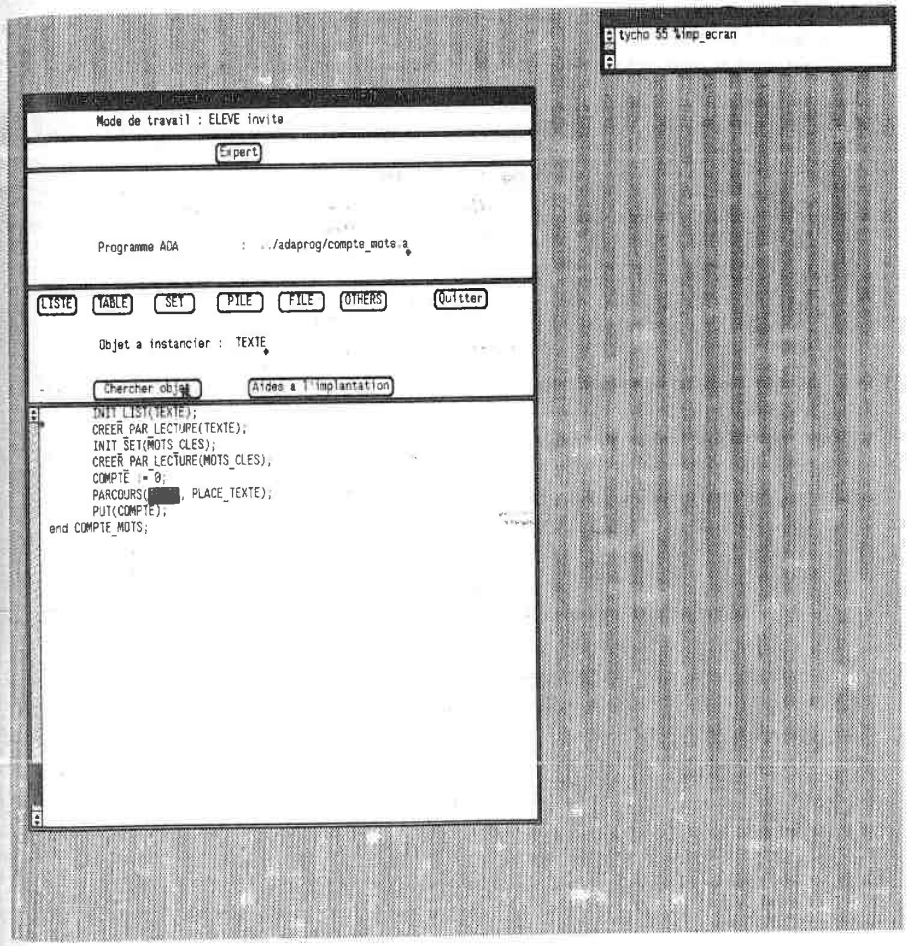


Figure 9.

Le type LISTE a été sélectionné, les autres types ne sont plus proposés. La liste des implantations que possède le système pour ce type est affichée; en caractères gras figurent ceux qui sont accessibles à l'élève "tintin" selon les droits que son enseignant lui a donnés, les autres sont nommés pour information et ne peuvent être sélectionnés.

Dans le texte de la procédure compte_mots, un choix d'implantation a déjà été fait pour la liste TEXTE (utilisation du package correspondant à une représentation chainée) et pour l'ensemble MOTS_CLES (représentation d'un ensemble par une liste chainée); on peut supposer que l'on veut ici modifier ces choix.

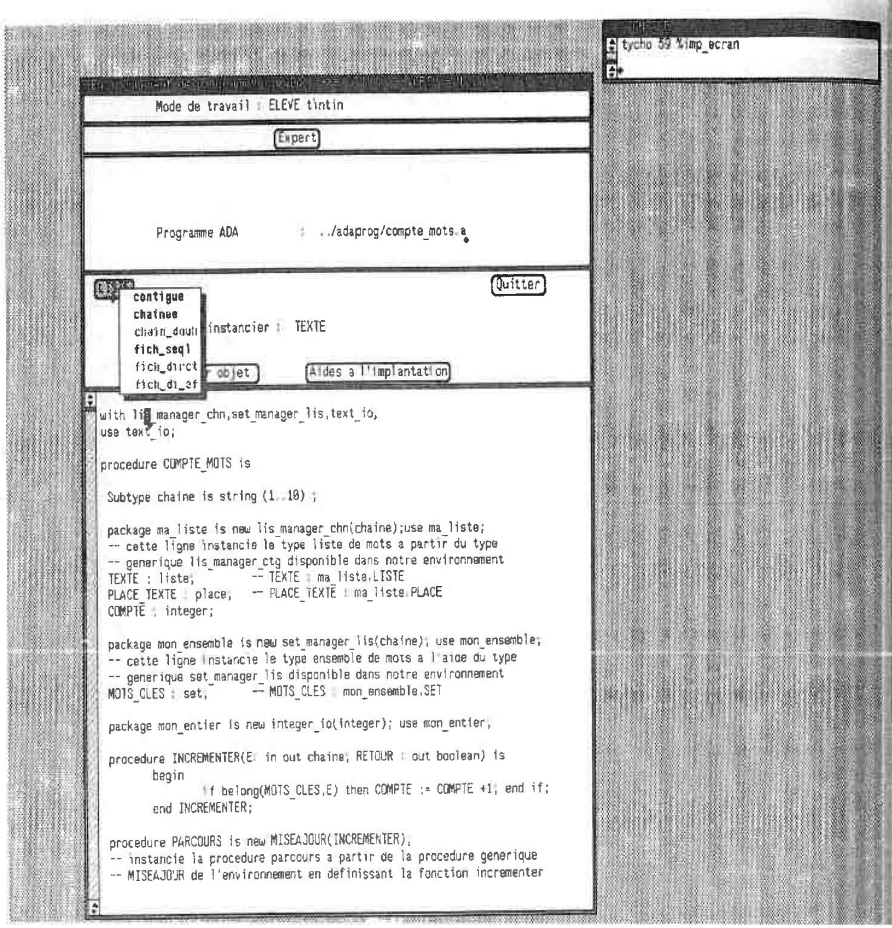


Figure 10.

L'élève a sélectionné EXPERT qui n'apparait plus en haut de la première fenêtre; la troisième fenêtre montre l'entrée dans le logiciel MORSE, suivie de la demande du nom de l'application.

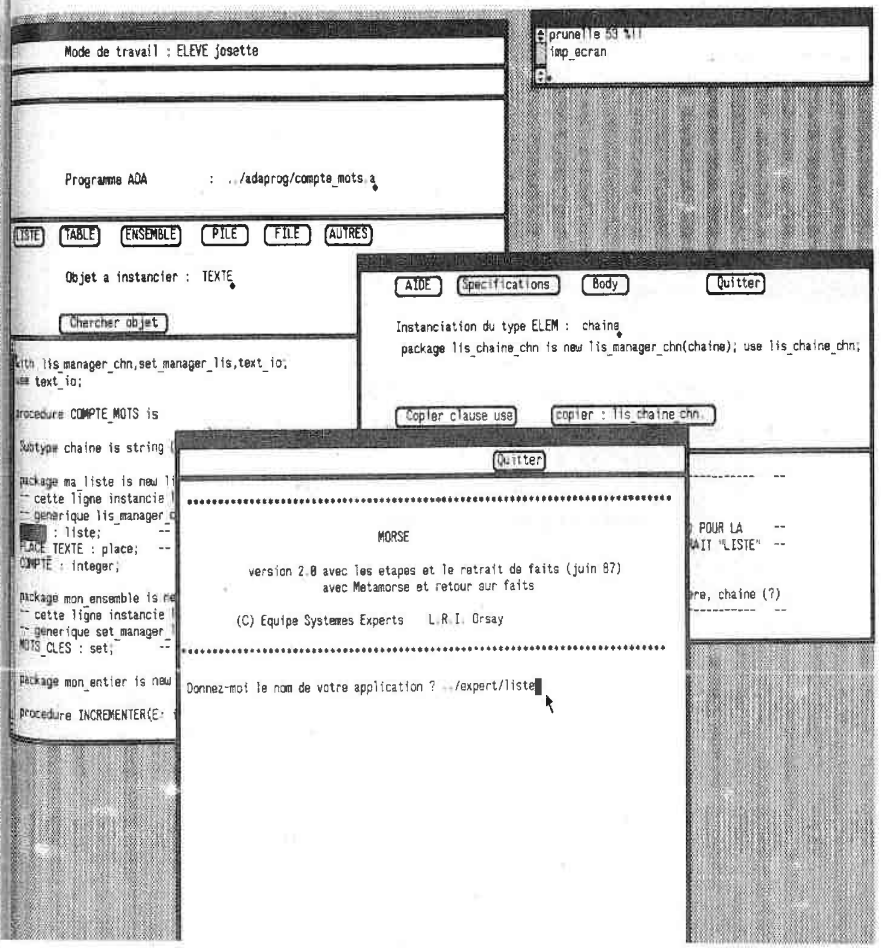


Figure 11.

Dans la troisième fenêtre, un dialogue se déroule à propos du choix d'implantation d'un objet de type LISTE, il a seulement valeur d'exemple, la base de règles ayant servi pour cette démonstration étant extrêmement réduite.

On notera pour chaque question la possibilité de faire afficher les valeurs attendues ou parfois la possibilité de ne pas répondre, ainsi que la sélection proposée du pourquoi qui amène le système à expliquer son raisonnement.

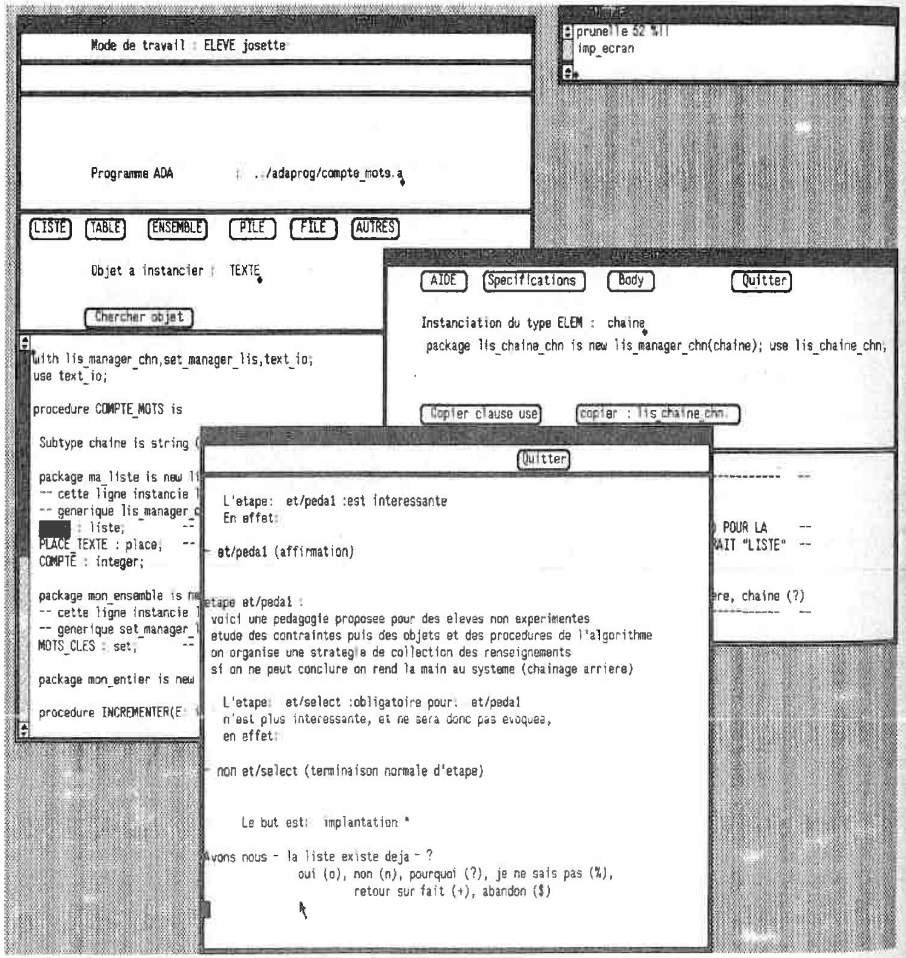


Figure 12.

L'élève a abandonné MORSE à l'aide du bouton QUITTER, a sélectionné une représentation chainée et demande dans la deuxième fenêtre une aide à l'implantation de cette représentation des listes. Il commence, à la demande du système, par préciser le type des éléments de la liste, ici CHAINE (de caractères).

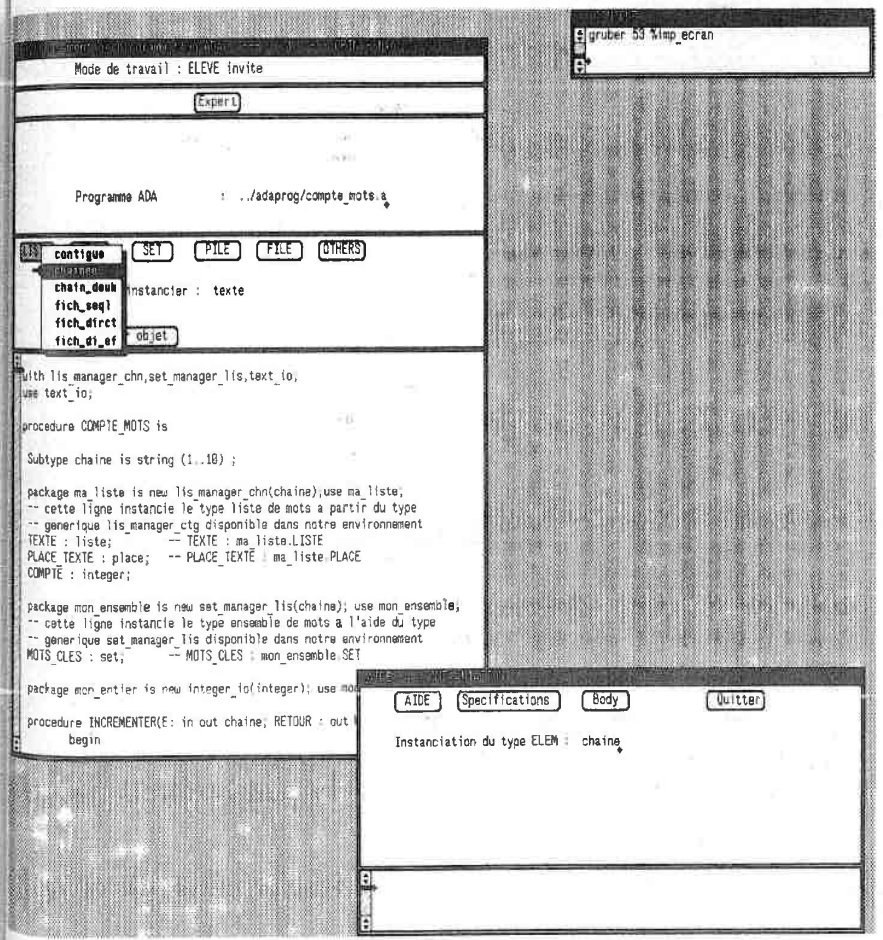


Figure 13.

Le système lui propose un texte instanciant le package de représentation chainée de liste; il peut, s'il le souhaite, utiliser ce texte, et, avec le bouton COPIER CLAUSE USE, l'insérer à l'endroit opportun de son texte ADA sur la première fenêtre. Le bouton COPIER lui permet de préfixer les noms d'identificateurs par le nom du package sans avoir à refrapper chaque fois la chaine de caractères correspondante.

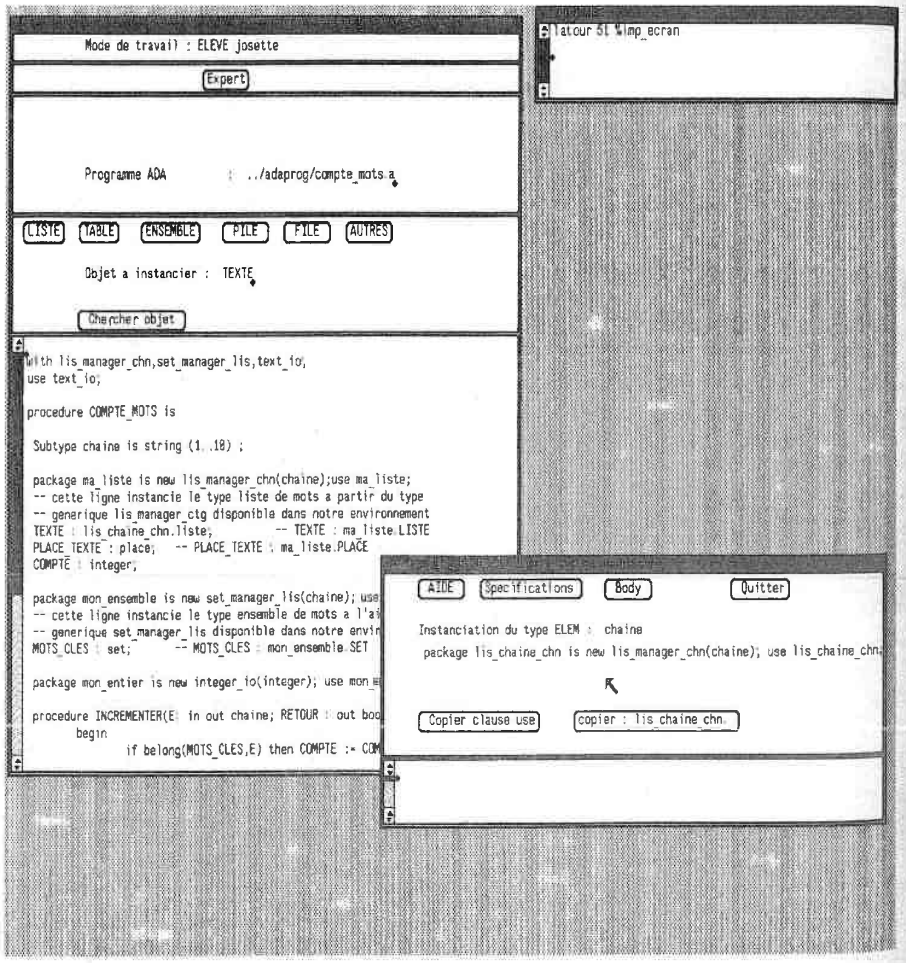


Figure 14.

La sélection du bouton SPECIFICATIONS permet d'afficher le texte des spécifications contenues dans la bibliothèque pour cette implantation du type LISTE.

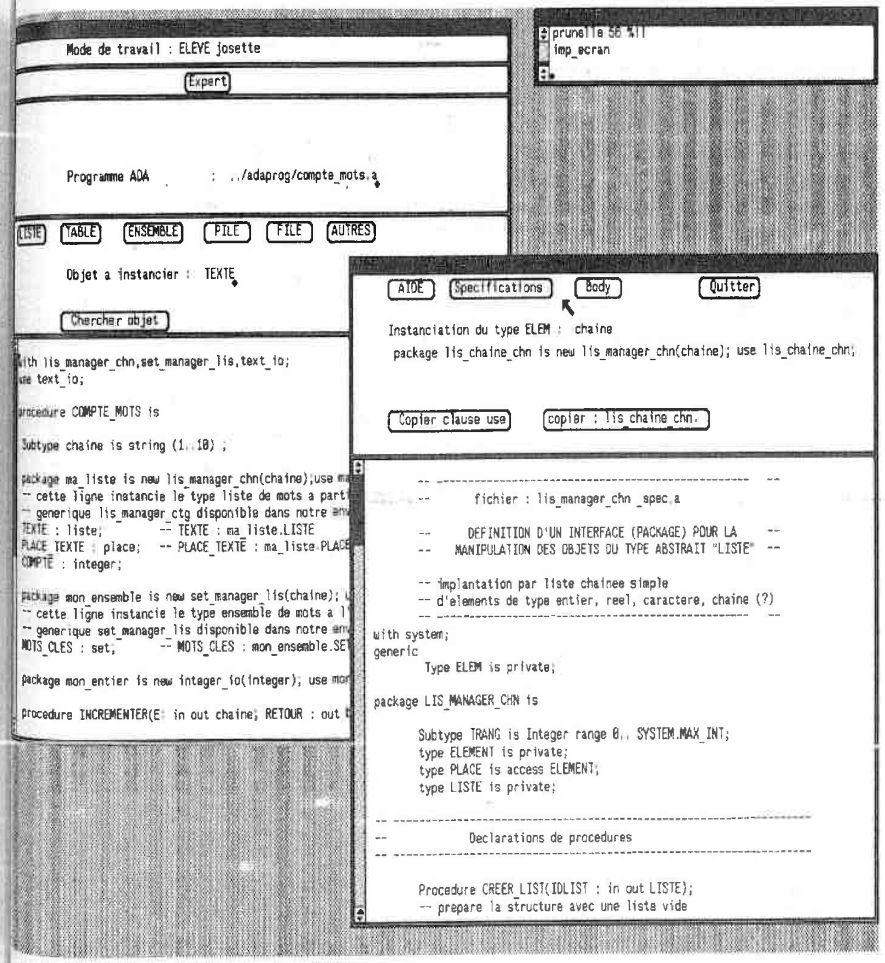
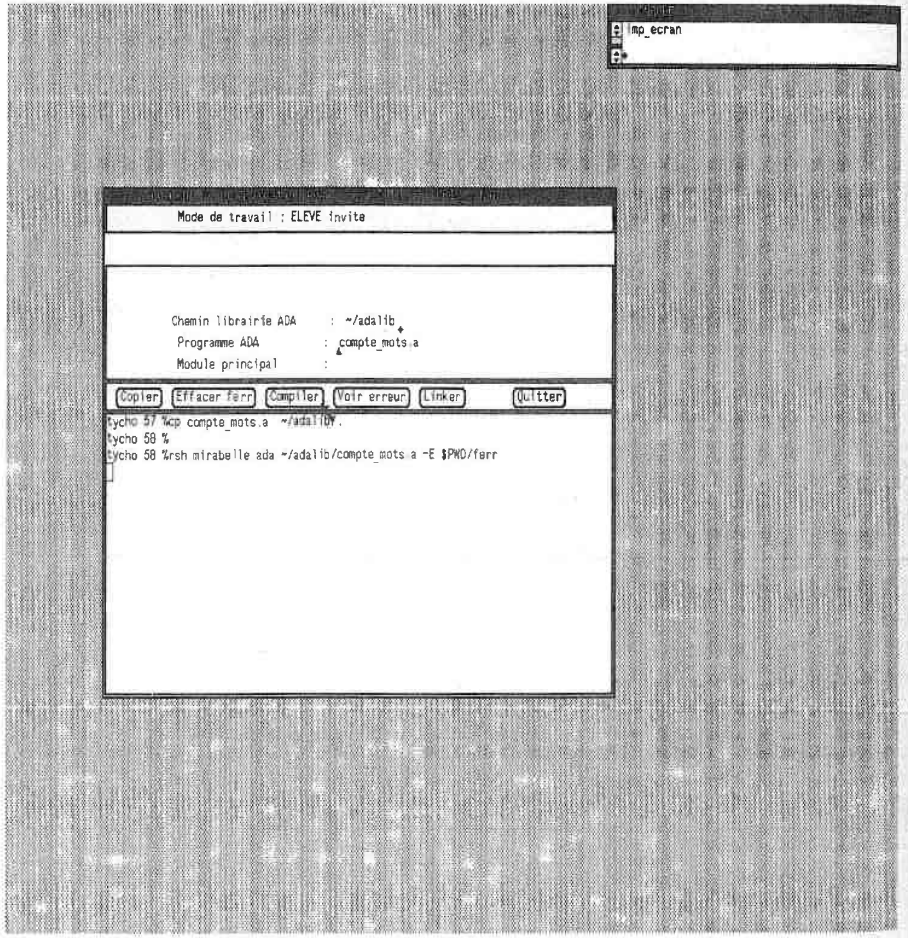


Figure 15.

L'élève est remonté dans l'arborescence des menus et veut maintenant compiler le programme qu'il a construit, le bouton COMPILER lui permet de générer automatiquement des fragments de commande UNIX fastidieuses à frapper, mais tout l'environnement UNIX est là disponible.

Il pourrait ensuite provoquer une édition de liens (LINKER), puis passer au chois EXECution dans le menu principal.



Chapitre 2

REALISATION DU SYSTEME SAIDA

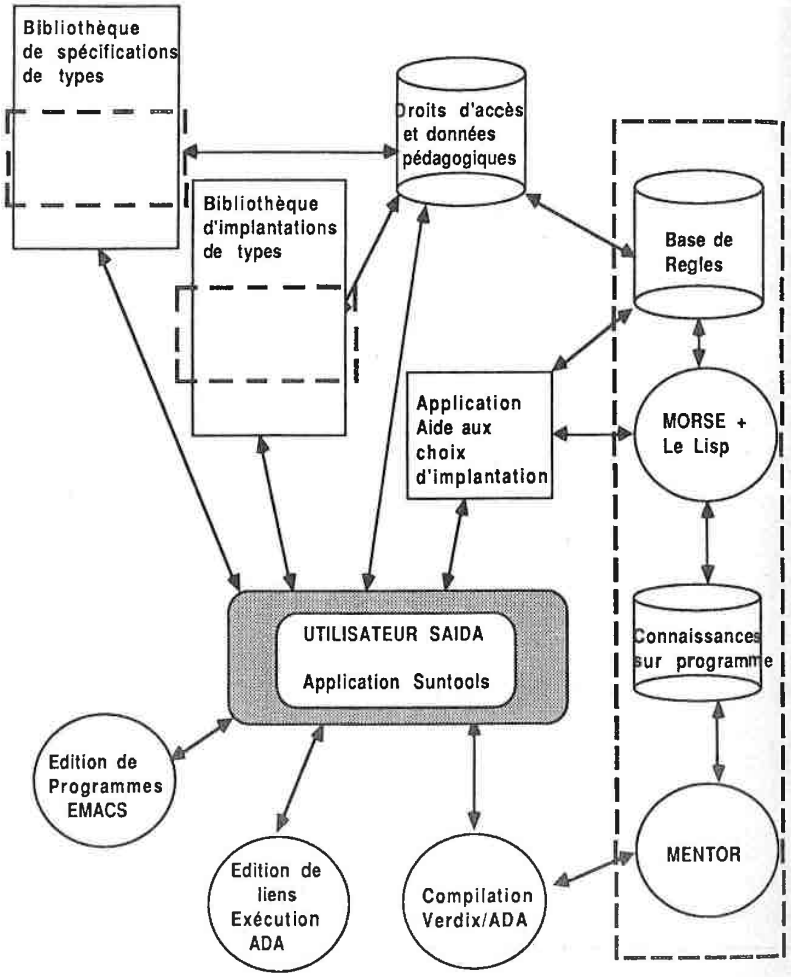
Le cahier des charges de SAIDA demande un système ouvert, interactif, appelant un moteur d'inférences, recherchant des informations à la fois dans le programme construit et auprès de l'utilisateur. Nous avons fait le choix de construire cet environnement à partir de logiciels existants ; cela nous semble être l'approche normale de réalisation d'une application chaque fois que les logiciels existants permettent de réaliser les fonctions souhaitées.

Dans ce chapitre, nous décrivons donc l'architecture générale du système, puis nous présentons les principaux logiciels avec lesquels l'application est réalisée. Pour piloter les interactions entre système et utilisateurs, nous avons choisi Suntools. Il nous fallait ensuite un générateur de système expert dans lequel on puisse regrouper des règles en étapes, MORSE, disponible sur Sun, écrit en Le Lisp et, a priori facilement portable malgré son caractère expérimental, a été retenu. Enfin, pour la recherche d'informations dans un programme, le laboratoire disposait de MENTOR/ADA ; nous n'avions pas besoin de toute la puissance de ce logiciel, son caractère interactif n'était pas le mieux adapté à une analyse qui reste cachée à l'élève ; il a été retenu en phase d'expérimentation.

Nous concluons sur les difficultés rencontrées pour faire coopérer les logiciels retenus.

2.1 ARCHITECTURE GENERALE DE SAIDA

SAIDA est une application disponible sur machine SUN/3 de MATRA Data System sous UNIX. C'est une application écrite avec Suntools qui fait appel à d'autres logiciels existants, à des programmes spécialement réalisés pour SAIDA, à des fichiers permanents et temporaires.



Les composants du système SAIDA

- Les logiciels utilisés

Un éditeur de textes pour la construction des programmes, TEXTEDIT de l'environnement Suntools.

Le gestionnaire d'écran Sunview.

Le compilateur VERDIX/ADA, l'environnement d'édition de liens et d'exécution associé.

L'interprête de LE LISP et le générateur de système expert MORSE écrit en LE LISP.

Le logiciel MENTOR/ADA.

- Les programmes écrits pour l'application

La gestion des dialogues par boutons, menus et multifenêtrage en Suntools.

La réalisation d'un mode utilisateur en LE LISP pour MORSE.

La recherche d'informations dans un programme ADA sous MENTOR.

- Les fichiers permanents

Les bases de règles formalisant le savoir du système en matière de choix d'implantations de données.

La bibliothèque de types implantée comme librairie ADA.

Les fichiers contenant les listes d'utilisateurs, leurs droits d'accès et les données pédagogiques.

- Les fichiers temporaires

Le fichier permettant de transmettre à MORSE les informations sur le programme issues de MENTOR/ADA.

Le schéma de la page précédente montre l'architecture générale du système.

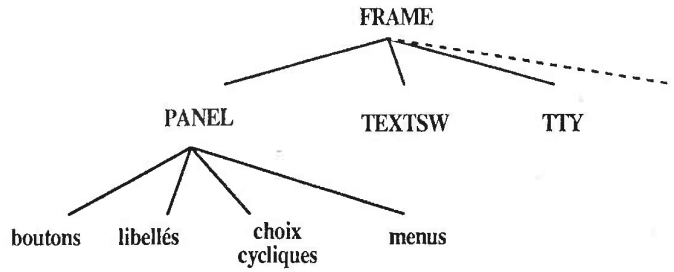
2.2 UNE APPLICATION INTERACTIVE REALISEE AVEC SUNTOOLS

L'application SAIDA est réalisée avec Suntools, un environnement d'écriture d'applications interactives construit au dessus de C et de UNIX.

2.2.1 Le cadre de définition d'une application

Sous SUNTOOLS, l'objectif d'une application est d'enchaîner certaines actions à partir de la gestion de l'espace écran. L'utilisateur définit en relation avec l'écran, les objets de son application et écrit le programme C qui leur assure le comportement désiré.

Pour cela, il a à sa disposition des catégories d'objets et des ensembles de fonctions attachées à chaque catégorie. Une catégorie principale est la fenêtre (Frame) ; elle peut être composée de sous-fenêtres de type panel, texte ou tty. Un panel peut contenir des libellés, des boutons, des menus, des choix cycliques, etc... . Une sous-fenêtre de type textsubwindow contient du texte, une fenêtre de type tty contient des commandes UNIX. Ces types d'objets sont structurés comme l'indique le schéma suivant :



Hiérarchie de catégories d'objets en Suntools

2.2.2 Exemple de mise en oeuvre

Les objets de l'application doivent être déclarés sous la forme :
< nom d'objet > : <catégorie Suntools>

Ils doivent ensuite être instanciés, c'est à dire être décrits par les propriétés qu'on leur affecte. Selon la catégorie de l'objet, ces propriétés concernent la localisation de l'objet sur l'écran, la donnée d'un libellé, l'activation d'une procédure C, etc...

Exemple : Description de l'objet pni_message

```
pni_message = panel_create_item
              ( pn_identité, PANEL_MESSAGE,
              PANEL_LABEL_STRING, "Donnez votre identité",
              PANEL_LABEL_Y, ATTR_ROW (2),
              PANEL_LABEL_X, ATTR_COL (10),
              0) ;
```

Ce qui signifie que l'objet pni_message appartient au panel de nom pn_identité, est de la catégorie PANEL_MESSAGE, a pour contenu "Donnez votre identité" et doit être écrit dans le panel en sautant deux rangées à partir du haut de l'écran et dix colonnes à partir du bord gauche.

Exemple : description de l'objet pni_student

```
pni_student = panel_create_item
              (pn_ada_choix, PANEL_BUTTON,
              PANEL_LABEL_IMAGE, panel_button_image (pn_ada_choix,choix, "Mode élève", 3, 0),
              PANEL_LABEL_X, ATTR_COL (1),
              PANEL_NOTIFY_PROC, student_proc,
              0) ;
```


L'objet pni_student appartient au panel pn_ada_choix, est de type bouton, contient le libellé "Mode élève" sur le bouton et si on clique sur ce bouton, la procédure écrite en langage C student_proc doit être activée.

2.2.3 Exemple de programme.

```
void copier_aide (item, event)      /* pour afficher le texte d'aide dans la fenêtre txt_specif */
        Panel_item ;                /* l'objet suntools associé est le bouton Aide */
        Event *event ;              /* l'évènement est un click gauche sur la souris */
{char nom_prog_a[70] ;              /* pour donner le chemin absolu du fichier contenant le texte */

if  (strlen (type_objet) != 0)      /* connait-on le type d'objet traité ? */
                                    /* si oui on génère le nom du fichier puis on le charge  */
        {sprintf (nom_prog_a, "%s/aide/%s_manager.aide",  getenv (saida),  type_objet) ;
        window_set (txt_specif, TEXTSW_BROWSING, FALSE,  0) ;
        textsw_reset (txt_specif, 0, 0) ;
        window_set (txt_specif, TEXTSW_FILE, nom_prog_a, TEXTSW_FIRST, 0, 0) ;
        window_set (txt_specif, TEXTSW_BROWSING, TRUE, 0) ;
        }                           /* si non on affiche un message d'erreur */
else    mess_bloque (pn_clause_use, event, "Selectionnez le type d'objet a implanter  n");
}
```

Suntools permet de programmer facilement des applications avec souris et multifenêtrage. Le type "fenêtre TTY" où l'on peut envoyer des libellés qui sont ensuite interprétés comme des commandes systèmes permet de lancer des programmes de façon totalement transparente à l'utilisateur.

Le texte qui réalise les fonctions décrites ci-dessus occupe 0,7 méga-octets.

2.3 UN LOGICIEL DE CREATION DE SYSTEME EXPERT : MORSE

Morse est un logiciel d'aide à la création de systèmes experts ; il se compose d'un langage d'expression des connaissances, d'un moteur d'inférences combinant mode déductif

et mode interactif de raisonnement. Il est décrit en Le Lisp 15.2 ; une partie du code, documenté dans la notice d'utilisation, reste accessible au programmeur qui peut ainsi ajouter certaines fonctionnalités ou modifier des libellés. Il a été écrit au LRI, Université Paris Sud, Centre d'Orsay dans l'équipe Intelligence artificielle et Systèmes d'inférences [KIR,85], [IASI,87] ; il est utilisé dans SAIDA dans le cadre d'échange de logiciels entre équipes participant au PRC-IA du CNRS. Il est disponible sur toute machine utilisant le système UNIX.

Nous reproduisons dans la suite du paragraphe quelques passages du manuel d'utilisation en les illustrant par des exemples empruntés aux bases de connaissances de SAIDA. Nous envisageons successivement la représentation des connaissances et la gestion de l'utilisation de ces connaissances.

Syntaxe

base de règles : liste de <règle>s
liste de <x> séparé par y correspond en notation Backus à : <x> [y <x>]*
liste de <x> ou <z> séparé par y ou w correspond en notation Backus à :
<x> | <z> [y | w <x> | <z>]*
liste de <x> séparé par y et terminée par w correspond en notation Backus à :
<x>[y <x>]* w
y,w sont des séparateurs ou des mots clés.
<règle> : nomderègle **si** <prémisse> **alors** <conséquent> .
rappel : nomderègle est une chaîne de caractères sans caractère blanc précédée par R et terminée par un point
<prémisse> : liste de <condition>s séparées par des virgules ou des SI
<condition> :
nomd'attribut comparateur nomd'attribut
nomd'attribut comparateur valeur
exemple : taille d'un élément = petite
nomd'attribut = liste de noms d'attribut(s) ou valeur(s) séparés par des OU
nomd'attribut (pour les attributs booléens)
non nomd'attribut (pour les attributs booléens)
exemple : non il y a des suppressions
(non peut être remplacé par pas)
nomd'attribut * (permet de savoir si cet attribut a une valeur dans la base de faits)
nomd'attribut ?? (correspond à la valeur "non obtenable" pour un attribut, est utilisé dans SAIDA pour les réponses "je ne sais pas")
<conséquent> : liste de <conclusion>s séparées par des virgules
<conclusion>

nomd'attribut = nomd'attribut
nomd'attribut = valeur
nomd'attribut
non nomd'attribut
nomd'attribut = <action>
<action>

Exemple : si le type de l'élément = entrée ou booléen ou réel ou caractère alors les éléments sont de type simple, la taille d'un élément = petite .

<action>: nom d'action (liste d'<argument>s séparés par des virgules)
rappel : nom d'action est une chaîne de caractères commençant par $ et ne comprenant pas de blanc.
<argument> :
nomd'attribut
valeur numérique : entier ou réel

2.3.1 La représentation de la connaissance

Une base de connaissances de MORSE se compose :
- d'une base de règles : les connaissances y sont exprimées sous forme de règles de production,
- d'une description de la structure du raisonnement ou étapes, celle ci est facultative,
- d'une base de faits, les faits sont des couples (attribut,valeur).

A. BASE DE REGLES : REGLES DE PRODUCTION

VOCABULAIRE

Il comprend :

- des mots-clés : **si, alors, non, pas, ou**
- des symboles spéciaux : ***, ??, :=**
- des comparateurs : **<, >, <=, >=, <>, =**
- des constantes appelées valeurs : entier, réel ou chaîne de caractères ; il existe deux valeurs spéciales : vrai et faux
- des noms d'attributs : chaîne de caractères
- des noms d'actions : chaîne de caractères sans caractère blanc et commençant par **$**
- des noms de règles : chaîne de caractères sans caractère blanc, commencant par **R** et

terminés par :
Exemple : Rdébut1:

- des séparateurs toujours précédés par des caractères blancs : (,), la virgule, le point.

Remarque sur le contrôle de types :
Un attribut peut être de type booléen, numérique ou chaîne de caractères. Le type d'un attribut est unique, il est calculé au moment de la compilation, l'utilisateur peut affecter un type à un attribut dans le fichier att-supp (voir options). Un certain nombre de contrôles sur les types sont effectués par le compilateur lors de l'analyse syntaxique.

LANGAGE D'INTERACTION

Une partie de la connaissance est entrée par l'utilisateur au cours de la session sous forme de <fait>s et de <but>s :

<fait> : nomd'attribut = valeur
nomd'attribut 1 = nomd'attribut 2
nomd'attribut (pour les attributs booléens)
non nomd'attribut (pour les attributs booléens) (peut être remplacé par **pas**)

<but > : <fait> **?**

B. BASE DE FAITS

La base de faits est constituée de :

- un ensemble de couples (attribut, valeur)
- une liste dite Liste-à-évaluer de couples (attribut1, attribut2).

(non utilisée dans SAIDA)

Tout ajout à la base de faits d'un couple (attribut, valeur) provoque la propagation du fait ajouté selon le mécanisme de chaînage avant.

La valeur d'un attribut est :

- val si le couple (attribut, val) appartient à la base de faits.
- non obtenable si le couple (attribut, ??) appartient à la base de faits.

L'attribut n'a pas de valeur sinon.

Les attributs sont monovalués.

C. STRUCTURATION DU RAISONNEMENT : LE FICHIER ETAPE.

Le fichier étape décrit la structure du raisonnement ; il est faculatif.

VOCABULAIRE

Il comprend :

- des mots-clés : **ETAPE, ENTREE, SORTIE, OBLIGATOIRE, ENVISAGEABLE, BUTS, COND-ARRET, FIN, $INSERTION-SORTIE, $INSERTION-FAITS**

Ces mots-clés seront écrits en majuscules et en gras dans la suite ; les lettres majuscules et minuscules sont traitées de la même manière par le compilateur.

- des symboles spéciaux : **. , ; ! et :**
- des noms d'étapes : chaîne de caractères
- des noms d'attributs : chaîne de caractères
- des valeurs : entier, réel ou chaîne de caractères

SYNTAXE

<fichier étapes> : liste d'<étape>s séparées par des points et terminée par **FIN.**

<étape> : **ETAPE** nomd'étape **:** liste de <entrée> ou <sous-étape> ou <sortie> ou <buts> ou <cond-arrêt> séparé par des points et terminée par un point.

<entrée> : **ENTREE** liste d'<impression> ou <entree-faits> ou **$INSERTION-FAITS**

<impression> : **!** chaîne de caractères quelconque
Tout ce qui est derrière le ! est considéré comme faisant partie de l'impresion et ceci jusqu'à la fin de la ligne.
Attention : le point terminant <entrée> ne doit pas être sur la ligne <impression>!
! seul imprime une ligne vide.

<entrée-faits> : liste de <fait> ou nomd'attribut **?**
<fait> : voir en A
<sortie> : **SORTIE** liste d'<impression> ou nomd'attribut ou **$INSERTION-SORTIES**
<sous-étape> : liste d'<étape-obligatoire> ou <étape-envisageable>
<étape-obligatoire>: **OBLIGATOIRE** liste de nomd'étapes
<étape-envisageable> : **ENVISAGEABLE** liste de nomd'étapes
<buts> : **BUTS** liste de <but>s
<but> : voir en A
<cond-arrêt> : **COND-ARRET** liste de <but>s

2.3.2 Le contrôle dans MORSE

MORSE travaille tantôt en mode déductif (chaînage avant), tantôt en mode interactif (chaînage arrière). Lorsqu'on ne fournit pas de fichier étape, les déductions ne sont déclenchées que par l'ajout de faits ou par une question proposant un but au système. Quand un ficher étape est proposé, l'étape "maîtresse" constitue le but proposé et contrôle l'enchaînement des raisonnements.

Le mode déductif

Pour toute règle R, si l'évaluation de la partie condition est vraie alors la partie conclusion est évaluée, ce qui en général entraîne l'adjonction de couples (attribut, valeur) dans la base de faits. Ce mode déductif fonctionne selon le principe dit de "saturation" : toutes les déductions rendues possibles à un instant donné sont effectuées.

Le mode interactif

Ce mode permet de poser des questions à l'utilisateur, notamment pour atteindre un but proposé lorsque le mode déductif n'a pas permis de conclure. Une question n'est posée que lorsqu'aucune règle n'a permis de conclure et lorsque l'attribut est demandable. C'est le concepteur du système qui fixe la liste des attributs demandables et non demandables.

Pour satisfaire un but, MORSE utilise une stratégie de généralisation. Si on lui demande "*la taille de l'élément = petite ?* ", avant de poser une question à l'utilisateur, il essaiera de trouver une autre valeur pour l'attribut taille de l'élément ; de même, s'il cherche à monter que vers-booléen = vrai, il tentera aussi de montrer vers-booléen = faux.

Le rôle des étapes

L'adjonction d'étapes permet de structurer le raisonnement en alternant des phase de chaînage avant déclenchées par un apport obligatoire d'informations par l'utilisateur et provoquant un ajout de faits dans la base de faits et des phases de chaînage arrière déclenchées par des buts et des sous-buts proposés.

A toute étape correspond un attribut dont le nom est celui de l'étape. Un attribut-étape peut avoir

- la valeur vraie qui représente l'étape "activée"
- la valeur faux qui représente l'étape "désactivée"
- pas de valeur

La valeur d'un attribut-étape peut être modifiée par l'évaluation des étapes ou par

l'évaluation des règles de production. Un attribut-étape peut aussi avoir la propriété " à envisager", donnée par des règles de production qui comporte l'action $envisager.

Une étape est caractérisée par un état. L'évaluation d'une étape (attribut étape à vrai) donne lieu successivement à l'évaluation

des entrées	état = entrée
des sous-étapes	état = corps
des sorties	état = sortie
des buts	état = fin

et fait passer l'attribut étape à faux.

Dans le corps d'une étape, une sous-étape peut être obligatoire ou envisageable. Les étapes obligatoires implantent dans SAIDA les noeuds ET de l'arbre de structuration des raisonnements. Les étapes envisageables implantent les noeuds OU et sont rendues activables selon les résultats des étapes précédemment activées par une action donnant la propriété "à envisager".

Pour une étude plus fine de l'outil MORSE nous renvoyons au manuel d'utilisation [KIR,85]. Les deux bases de règles et les fichiers étapes réalisés sont donnés en annexe.

2.4 RECHERCHE D'INFORMATIONS DANS UN PROGRAMME : MENTOR - ADA

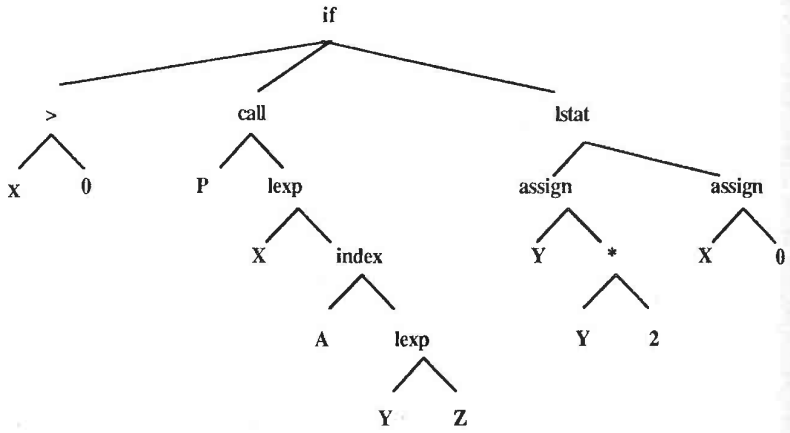
Comme nous l'avons expliqué au chapitre 2 de la troisième partie, certaines des informations dont le système à base de connaissances a besoin pour proposer des implantations d'objets peuvent être extraites automatiquement du texte de programme ADA. Mentor est un éditeur syntaxique interactif mis au point à l'IRIA [DONZ,80] ; la section Mentor-Ada permet de manipuler des programmes ADA représentés par les arbres de syntaxe abstraite associés. Pour programmer des recherches d'informations dans ces structures, l'utilisateur dispose du langage de manipulation d'arbres MENTOL.

2.4.1 Exemple d'arbre abstrait

Soit le texte ADA suivant :

if X > 0 then P (X, A (Y, Z)) ; else Y : = Y*2 ; X :=0 ; end if ;

MENTOR/ADA manipule l'arbre abstrait associé suivant :



On trouve aux noeuds de cet arbre des éléments d'un ensemble de sortes caractérisant le langage (call = appel, lex = liste d'expression, lstat = liste d'instruction...) et des éléments d'un ensemble d'opérateurs.

2.4.2 Mentor - Ada et SAIDA

Nous utilisons Mentor-Ada pour obtenir sur la procédure ADA analysée des informations relevant d'une analyse statique ; nous recherchons dans une procédure :
- les déclarations d'objet de certains types,
- pour chaque objet déclaré, la liste et le niveau d'imbrication des procédures prédéfinies dans la spécification du type qui utilisent cet objet.

Les procédures décrites ci-après fournissent ces résultats ; elles ne sont pas intégrées au système dans sa version actuelle. En effet, MENTOR est un éditeur interactif et nous souhaitons l'utiliser sans aucune intervention de l'utilisateur ... Il faut constituer un fichier de demandes issues du logiciel MORSE (lui aussi interactif !) et restituer à celui-ci les résultats des procédures MENTOL écrites.

2.4.3 Procédures implantées

Quelques procédures ont été écrites avec l'unique objectif de montrer la faisabilité de ce type d'analyse. Les pages suivantes donnent les algorithmes utilisés, des exemples de procédures MENTOL traduisant ces algorithmes et le résultat de l'application de ces procédures à un texte ADA.

Algorithme de traitement d'un arbre syntaxique issu d'un programme ADA

```
Module GO
Initialiser le fichier_des_résultats
CHERCHE_DECLARATIONS
Fermer le fichier_des_résultats

Module CHERCHE_DECLARATIONS
Se placer dans la partie déclaration du programme principal
Pour  chaque déclaration du programme principal
faire  CHERCHE_LISTES
finpour

Module CHERCHE_LISTES
Si     c'est une déclaration de variable(s)
alors  si      c'est une déclaration d'objet(s) de type LISTE
       alors   pour   chaque objet_déclaré
               faire   TRAITE_UNE_LISTE
               finpour
       finsi
finsi

Module TRAITE_UNE_LISTE
pour  chaque procédure prédéfinie
faire  initialiser une liste de compteurs
       mémoriser le nom de la procédure prédéfinie traitée
       initialiser un compteur cpt
       procédure_traitée = prodédure_prédéfinie
       se placer sur le corps du programme
       EXAM(objet_declaré,le_corps_du_programme,cpt)
finpour
pour  chaque procédure_générique_prédéfinie
faire  TRAITE_PROC_GEN(procédure_générique_prédéfinie)
finpour

Module TRAITE_PROC_GEN
initialiser une liste de compteurs
mémoriser le nom de la procédure générique traitée
pour  chaque déclaration du programme principal
faire  si déclaration = instanciation de procédure générique
       alors   si procédure_générique_citée = procédure_générique_prédéfinie
               alors   TRAITE_PROC
               finsi
       finsi
finpour

Module TRAITE_PROC
procédure_traitée = procédure_instanciée
initialiser un compteur cpt
se placer sur le corps du programme
EXAM(objet_declaré,le_corps_du_programme,cpt)

Module CHERCHE_PARAM
       Si      position_eff = param_eff
       alors   paramètre formel= position_form
       sinon   avancer dans la liste des paramètres formels
               avancer dans le liste des paramètres effectifs
               CHERCHE_PARAM(nouv_pos_eff,nouv_pos_form,param_eff)
```

```
Module EXAM
 Pour  chaque instruction de la liste à examiner
 faire  EXAMINE(objet,instruction,compteur)
 finpour
 Si     compteur <> 0
 alors  adjonction(liste des compteurs, compteur)
 finsi

Module EXAMINE
 Cas instruction est un appel de procédure  alors
        si      c'est la procédure traitée
        alors   si      c'est l'objet examiné
                alors   incrémenter le compteur
                finsi
        sinon   AUTRE_PROC
        finsi
 Cas instruction est une itération alors
        créer un nouveau compteur
        incrémenter le niveau d'itération
        EXAM(objet,la liste des instructions itérées, nouveau_compteur)
 Cas instruction est une conditionnelle alors
        créer un nouveau compteur
        EXAM(objet,liste d'instructions de la partie alors, nouveau_compteur)
        créer un nouveau compteur
        EXAM(objet,liste d'instructions de la partie sinon, nouveau_compteur)
 Cas instruction est un cas alors
        pour    chaque cas étudié
        faire   créer un nouveau compteur
            EXAM(objet,liste d'instructions correspondante, nouveau_compteur)
        finpour
 fincas

Module AUTRE_PROC
 Si     l'objet est un paramètre effectif de l'appel
 alors  si      la forme du paramètre effectif est une association
        alors   si      le fils droit = objet
                alors   paramètre_formel = fils gauche
                        chercher le nom de la nouvelle_procédure
                        créer un nouveau compteur
                        aller sur la partie déclaration du programme
                        chercher la déclaration de la nouvelle procédure
                        EXAM(paramètre_formel,liste d'instructions de la
nouvelle procédure,nouveau_compteur)
                finsi
        sinon
                chercher le nom de la nouvelle_procédure
                aller sur la partie déclaration du programme
                chercher la déclaration de la nouvelle procédure
                position_eff = début de la liste des paramètres effectifs
                position_form = début de la liste des paramètres formels
                CHERCHE_PARAM(position_eff,position_form,param_eff)
                créer un nouveau compteur
                EXAM(paramètre_formel,liste d'instructions de la nouvelle
procédure,nouveau_compteur)
        finsi
 finsi
```



```
.rec
mentol
% ------------------------------------------------------------------------------
% definitions de commandes d'analyse de texte de programmes avec listes
%                       en      ADA
% programme principal d'analyse pour un objet declare L de type liste
:&
.redef<.go,(
                .devout<@f_analise>;
                .top;
                .cherche_decl;
                .devout;
        )>

ss2;.lredef
% ------------------------------------------------------------------------------
%       cherche_decl: recherche de declaration d'objets de type liste
%               dans la partie declaration du bloc examine
%               place le nom de l'objet dans une variable globale @ma_liste
:&
.redef<.cherche_decl,
                (ff@subprogram_body/
                 (f@item_s/     % va sur la partie declaration du prog princ
                  ?.apl<.cherche_listes>,.cherche_listes;);
                )>

ss2;.lredef
% ------------------------------------------------------------------------------
%       cherche_listes : recherche d'une declaration de listes
%               @objet est une variable globale qui contient le nom de la liste
%               examinee
%               @marque_decl: variable locale marque la declaration examinee
%               @marque_liste: variable locale marque l'objet liste examine
:&
.redef<.cherche_listes<@marque_decl,@marque_liste>,
        (.is<@var>/     %seules les declarations d'objets sont examinees
                (@marque_decl:;         %marque la declaration qu'on traite
                f@type_liste            %cherche si declare type liste
                /(u2; s1;               %remonte  si oui va a gauche
                  .is<@id_s>;   %est-ce une liste d'identificateurs?
                      ?(.apl<(@marque_liste:;   %traite chaque identificateur
                                @objet = @k;
                                @objet p;       %dans le fichier sortie
                                .traite_une_liste<@k>;
                                :@marque_liste;)> ),
                      (@objet = @k;
                       @objet p;
                       .traite_une_liste<@k>;);%traite un seul identificateur
                 );
                :@marque_decl;
                );
        )>

ss2;.lredef
^
```

Exemple de programme ADA soumis à l'analyse

```
 -- compilation
        -- comp_unit
                --context
                -- subprogram_body
procedure ADA is

                        -- item_s
        type    liste is array(1..5) of integer;  -- type
        valeur:integer:=0;                        -- var
        l1,l2:liste;                              -- var
        i,j : integer;
        l3 : liste;
        l4 :liste;              -- (declaration inutilisee)
        l5 : liste;

        procedure trou(i,k: in integer; l:in out liste) is -- subprogram_body
        begin
        k:=0;
        for ii in 1.. 10 loop
                init_list(l);
                end loop;
        k:=1;
        end trou;

        procedure compter is                      -- subprogram_body
        begin
                null;
        end compter;
        procedure parcours is new parc_total(compter);

                                                  -- partie stm_s
begin
        init_list(l1);                            -- call          -- 1
        j:= 0;                                    -- assign        -- 2
        init_list(l1);                                             -- 3
        init_list(l2);                                             -- 4
        for ii in 1..4  loop                      -- loop          -- 5
                j:= j+1;                                           -- 6
                init_list(l1);                                     -- 7
                end loop;
        init_list(l3);                                             -- 8
        if i>j then                               -- if            -- 9
                j:=0;                                              -- 10
                init_list(l1);                                     -- 11
                j:= 1;                                             -- 12
        else    j:=5;                             --cond_clause    -- 13
                init_list(l2);                                     -- 14
        end if;
        for ii in 1..3  loop                      -- (double iteration) -- 15
                j:=1;                                              -- 16
                for jj in 2.. 3 loop                               -- 17
                        init_list(l1);                             -- 18
                        end loop;
                end loop;                                          -- 19
        init_list(l1);
        trou(i=>i,k=>j,l=>l5);                    -- (appel de procedure) -- 20
        parcours(l5);                             -- (procedure generique)-- 21
        for ii in 1..5 loop                                        -- 22
                parcours(l5);                                      -- 23
                end loop;
end ADA;
```


Exemple d'analyse des objets l1, l2, l3, l4, l5

```
l1                      -- premiere variable de type liste
init_list;              -- examen de la procedure init_list
niveau := v3;           -- instructions 1,3,19
niveau2 := v1;          -- 18
niveau := v1;           -- 11
niveau1 := v1;          -- 7
parc_total;
l2
init_list;
niveau := v1;           -- 4
niveau := v1;           -- 14
parc_total;
l3
init_list;
niveau := v1;           -- 8
parc_total;
l4
init_list;
parc_total;
l5
init_list;
niveau1 := v1;          -- 20
parc_total;
niveau := v1;           -- 21
niveau1 := v1;          -- 23
```

Chapitre 3

CONSTRUIRE DES ENVIRONNEMENTS D'ENSEIGNEMENT AUTOUR DE SYSTEMES A BASE DE CONNAISSANCES

Notre objectif dans ce chapitre est de tirer des enseignements de la construction de SAIDA et de proposer des recommandations plus générales pour la construction d' environnements d'enseignement autour de systèmes à base de connaissances.

Des systèmes experts sont aujourd'hui opérationnels sur des sites de production industrielle, dans les banques, les administrations, les centres de recherche [RAU,83], [REIT,84], [ERN,85], [BENCH,86], [GRAN,87] ; leur mise au point a nécessité la constitution de bases de connaissances importantes et la mobilisation de nombreux experts des domaines concernés. Ces bases peuvent, sauf dans certains cas de confidentialité, être utilisées à des fins d'enseignement.

Nous avons vu que les bases formalisant les connaissances d'un domaine d'application particulier étaient l'une des composantes fondamentales des "tuteurs intelligents" qui devraient constituer la génération future des environnements d'enseignement assisté par ordinateur.

M. Baron a indiqué dans [BARON,84] quelques propriétés que devait posséder une base de connaissances pour être utilisée à des fins pédagogiques, à partir des propositions faites par Clancey dans [CLAN,81]. Les travaux de Clancey se poursuivent autour de GUIDON 2 [CLAN,87]. Nous prolongeons ici ces réflexions en expliquant quels sont les facteurs qui, à notre avis, plaident pour une utilisation de bases existantes, quels sont ceux qui appellent la création de bases spécifiques et en indiquant dans chacun des cas un certain nombre de conditions techniques ou méthodologiques à respecter pour y parvenir.

Par ailleurs, la construction ou la modification de bases de connaissances pour l'enseignement pose la question de savoir qui construit ou qui modifie et quels outils on fournit pour ce type d'activité.

3.1 UTILISER DES BASES EXISTANTES

Des facteurs plaident pour la réutilisation de bases existantes, citons les principaux :

- Construire une base de connaissances concernant un domaine d'application significatif demande un investissement humain important ; pour les bases existantes, cet investissement a été réalisé, autant l'utiliser.

- Ces bases sont faites pour des professionnels, elles contiennent les connaissances au top niveau de la spécialité, il est intéressant de mettre ces connaissances professionnelles à la disposition de ceux qui apprennent. De telles bases existent dans des domaines variés, comme le dépannage, la chimie, les mathématiques, la gestion financière, la médecine, etc...

Mais, il y a des conditions à remplir :

- Globalement une telle utilisation n'est possible que si on fait, au niveau cognitif, l'hypothèse que le savoir de l'élève est un sous-ensemble de celui de l'expert, sous-ensemble que l'élève a pour objectif d'accroître.

- La base de connaissances doit être structurée à la fois sur le plan de l'univers de travail et sur celui des règles exprimant les raisonnements, et si possible structurée selon de multiples points de vue. Nous avons vu dans notre application toute l'importance de la structuration de l'univers pour construire les dialogues pédagogiques proposés au chapitre 3 de la seconde partie. Les structurations proposées doivent être celles sur lesquelles on veut s'appuyer dans la démarche pédagogique prévue.

- Il faut disposer d'outils d'expression de connaissances pédagogiques aussi puissants que ceux disponibles pour les connaissances du domaine.

- Il faut prévoir une utilisation par "couches" ou par "thème" de cette base pour un apprentissage progressif des concepts et techniques du domaine ; dans SAIDA, il faudrait par exemple exprimer facilement le retrait provisoire d'une représentation physique et travailler avec une version de la base ne contenant pas cette représentation mais restant cohérente

[AYE,86]. Une telle fonctionnalité nous semble essentielle dans des systèmes d'enseignement pour permettre au professeur de fabriquer à partir de la base professionnelle, celle qu'il met à la disposition des élèves. Dans les systèmes à finalité professionnelle, les études ont plutôt porté sur l'adjonction de connaissances et le maintien de la cohérence de la base à cette occasion, mais la question de la mise à jour d'une base de connaissances par retrait d'éléments devenus périmés doit aussi être résolue. On peut sans doute s'inspirer de la façon dont cette question a été abordée pour les bases de données et déterminer à partir de là des caractères spécifiques aux bases de connaissances appelant des solutions à inventer.

Lorsque ces conditions sont remplies, on peut entreprendre la conception et la réalisation de couches pédagogiques autour de telles bases. Mais il y a aussi des facteurs qui empêchent la réutilisation d'une base existante. Il faut alors s'orienter vers la construction de bases nouvelles.

3.2 CONSTRUIRE DES BASES NOUVELLES

Quand faut-il prendre cette décision ?

- Lorsque les bases existantes ne sont pas correctement structurées ou n'utilisent pas les concepts que l'on souhaite introduire.

Il faut remarquer ici que les bases de connaissances conçues à des fins professionnelles s'adressent à un public souvent homogène, formé de spécialistes connaissant bien leur domaine, notamment au niveau du vocabulaire, de l'environnement, etc... . L'utilisation à des fins pédagogiques suppose un public varié, se familiarisant avec le domaine, qui peut ne pas "comprendre" la connaissance du système si elle n'est pas reformulée en tenant compte de ces caractères propres.

Il s'agit alors de refaire une base, à partir d'une expertise déjà formulée.

- Lorsque les stratégies de raisonnement ne sont pas pertinentes et ne correspondent pas aux contenus pédagogiques prévus ; il suffit alors de refaire une partie de la base, celle consacrée à la conduite des raisonnements.

- Il n'existe pas de base dans le domaine considéré, c'est le cas de l'enseignement de la géométrie en collège par exemple [CHO,87a&b].

Des conditions à remplir

- Choisir le ou les formalismes les mieux adaptés pour représenter les connaissances du domaine : la nature des connaissances varie d'un domaine à l'autre, une étude fine des caractéristiques de la connaissance dans chaque domaine doit être menée avant de faire un choix.

- La construction d'une base est un investissement important ; si on le fait avec un objectif pédagogique, il faut prévoir son utilisation pour des publics élèves différents et des stratégies pédagogiques d'enseignants différentes avec de larges possibilités de paramétrisation.

A mi-chemin entre l'utilisation professionnelle et la formation se trouve la formation professionnelle. Dans ce cadre, l'élève connait de mieux en mieux les concepts du domaine, il doit apprendre les raisonnements pointus et les outils des professionnels de ce domaine. C'est donc dans ce secteur que les bases professionnelles peuvent le plus facilement être utilisées en formation. On y voit d'ailleurs de plus en plus d'applications spécialement dédiées à la préparation au poste de travail [HAT,88] ou à l'aide au pilotage en situation normale ou accidentelle [EDF,88].

En formation générale, il faut probablement développer des systèmes spécialement conçus pour l'enseignement. Des outils généraux de conception de systèmes d'EIAO, qui faciliteraient ces développements font certes défaut, malgré des tentatives actuelles en ce sens [REG,87], [REG,88], [VIV,87a], [PAL,88] ; mais nous pensons que si de tels outils étaient rapidement disponibles, et la transposition actuelle sur micro-ordinateurs PC de générateurs de systèmes à bases de connaissances comme KOOL ou ART va permettre de construire rapidement de tels outils, il resterait un gros travail de formalisation des domaines d'enseignement et des stratégies de raisonnement à enseigner dans ces domaines.

Quelques réalisations significatives sont en cours pour la géométrie, l'allemand, le latin, la gestion, le droit [CLE,87]. D'autres travaux réalisés à l'étranger méritent aussi attention, mais l'enseignement est un domaine d'application de l'informatique dans lequel il est plus difficile d'importer que dans d'autres ; il véhicule en effet une part d'identité culturelle nationale qui doit savoir élargir ses horizons tout en préservant ses traditions.

Un tel travail de conception et de production de bases de connaissances, qui ne peut être fait que par des pédagogues, est long à accomplir ; c'est pourquoi nous pensons qu'il faut le commencer avec les outils d'aujourd'hui [GRAN,84]. C'est le sens de notre démarche avec SAIDA pour l'enseignement de l'informatique.

Mais, il faudra penser dans les systèmes de demain à la collaboration entre ceux qui concevront les bases de connaissances, et c'est un travail trop important pour que chaque enseignant s'y mette, et les utilisateurs de ces bases qui voudront légitimement conserver un espace de liberté. L'expert du domaine, l'expert en didactique de la discipline sont des acteurs qui devront collaborer avec le professeur pour définir les environnements d'apprentissage à mettre à la disposition des élèves.

Conclusion

CONCLUSION

Nous avons indiqué dans l'introduction que cette recherche se situait au carrefour de trois domaines de l'informatique :

- la construction de programmes,
- l'intelligence artificielle,
- l'informatique et l'enseignement.

Elle avait pour objectifs la formalisation et l'expression des raisonnements à mettre en oeuvre pour représenter efficacement des objets abstraits dans un langage de programmation impératif, la réalisation d'un environnement d'enseignement mettant une telle expertise à la disposition des élèves et leur permettant de se l'approprier et enfin une réflexion sur la conception et la réalisation de tels environnements.

Au terme de sa présentation, nous voulons porter un regard critique sur l'ensemble , c'est à dire mettre en évidence les principaux apports de ce travail, mais aussi en souligner les limites et surtout indiquer les directions nombreuses et variées dans lesquelles il pourrait être poursuivi. Notre bilan porte essentiellement sur les idées et les concepts mis en oeuvre ; nous attachions de l'importance à la production d'une maquette, mais nous sommes tout à fait conscients du caractère inachevé de la réalisation actuelle et des améliorations qu'elle nécessiterait, notamment sur le plan ergonomique, pour devenir véritablement opérationnelle.

Nous distinguons entre le sujet de notre étude, la construction de programmes et son enseignement, et les méthodes et outils de construction de bases de connaissances et

d'environnements à finalités pédagogiques. Chaque fois que c'est possible, nous essayons de faire apparaître, d'une part les études fondamentales sans lesquelles il n'y aura pas, à notre sens, d'avancées notables dans les secteurs concernés et les logiciels qui apparaîssent nécessaires pour aboutir plus rapidement à des applications fiables, efficaces et conviviales.

Cependant, toute classification nous apparaît arbitraire, tant ce travail nous a conduits à emprunter des méthodes, des concepts et des outils à plusieurs domaines de l'informatique ; aussi commençons-nous ce bilan par une rapide description des caractéristiques d'une telle démarche et des difficultés qu'elle comporte.

Une recherche au carrefour de plusieurs domaines de l'informatique

Les travaux interdisciplinaires sont rares dans le monde de la recherche et pourtant, dès qu'on quitte les exercices d'école pour aborder les problèmes réels, il apparaît que toute recherche de solutions fait appel à des connaissances issues de domaines variés. On retrouve ce cloisonnement entre les domaines d'une même discipline. L'enseignement est certainement un thème fédérateur privilégié pour des travaux de ce type, notamment en informatique ; M.Quéré avait ouvert la voie en 1980 [QUE,80] avec une contribution à l'amélioration des processus d'enseignement et d'organisation de l'éducation, considérant l'ordinateur à la fois comme outil et objet de formation. Nous pensons l'avoir poursuivie en nous limitant aux questions d'enseignement et d'apprentissage, en utilisant largement les concepts et techniques de l'intelligence artificielle tant sur le plan de la matière enseignée, la programmation, que sur sur celui de l'assistance "intelligente" offerte par le système et en choisissant comme thèmes enseignés des concepts et techniques de programmation avancés.

La confrontation de concepts, méthodes et outils de domaines différents dans la réalisation d'une application renvoie très vite à de nouvelles interrogations dans chacun de ces domaines ; et d'un certain point de vue, nous avons soulevé beaucoup plus de questions que nous n'en avons résolu. Une approche de cette nature a aussi ses limites par rapport à des études plus spécialisées. Nous en soulignons deux :

- sur le plan des contenus, les sujets abordés sont nombreux et ne peuvent être approfondis chacun comme ils le mériteraient,

- sur le plan de la présentation du travail, on s'adresse nécessairement à des lecteurs de spécialités différentes ; il faut présenter à chacun l'essentiel du domaine qui ne

lui est pas familier sans tomber dans l'encyclopédisme ; entre les deux, la voie est étroite et la tentation grande de tomber d'un coté ou de l'autre. Nous avons essayé de bien décrire les différents points de vue selon lesquels notre problème pouvait être abordé dans la première partie de l'ouvrage. Ensuite, nous avons rappelé les définitions qui nous semblaient essentielles ; que ceux qui ont trouvé leur lecture fastidieuse se souviennent qu'elles étaient destinées à des non-spécialistes ; les références bibliographiques nombreuses devraient pouvoir satisfaire la curiosité de ceux qui, au contraire, auraient jugé notre exposé trop rapide sur d'autres points.

La construction de programmes et son enseignement

Un système tel que SAIDA nous semble avoir de l'intérêt à la fois pour la formation de spécialistes de la programmation, mais aussi pour le développement des environnements de programmation assistée que l'on devrait trouver sur les stations de génie logiciel de demain.

Sous l'angle de la construction de programmes

Notre réalisation peut être analysée sous l'angle de la construction assistée de programmes. Elle permet, en effet, de passer d'une procédure utilisant des types abstraits de données, empruntés à une bibliothèque prédéfinie, à un programme exécutable, comprenant une implantation efficace pour chacun des objets abstraits de la procédure. A notre avis, les environnements de programmation dotés de bibliothèques de composants ne seront réellement opérationnels que lorsqu'ils offriront à l'utilisateur des outils d'assistance à la recherche et au choix de composants. C'est l'idée maîtresse de SAIDA qui apporte une bibliothèque de composants relatifs aux implantations de types abstraits de données et une aide au choix de composants pour les applications définies par l'utilisateur.

Sous cet angle, les principaux apports de notre travail concernent :

- l'expression, dans une base de règles, de critères de choix qui conduisent à des propositions d'implantations efficaces, l'ensemble étant largement indépendant du langage support,

- une bibliothèque de composants logiciels à utiliser en liaison avec la base de règles précédente ou séparément.

Il pourrait être poursuivi dans les directions suivantes :

a) en conservant les choix actuels

- augmentation de la bibliothèque de types (davantage de types, davantage d'opérateurs et de représentations pour un type),

- augmentation de la base de connaissances de la même façon que la bibliothèque, mais surtout prenant en compte des critères d'analyse d'algorithmes plus fins que ceux utilisés aujourdhui. Il faudrait notamment mieux étudier les compositions de types (listes de tables, ensembles de listes,...) et traiter globalement la question de la représentation de tous les objets d'un programme alors que chacun d'eux est pour l'instant envisagé séparément avec seulement quelques références à l'environnement.

b) en recherchant plus de généralité ou de proximité d'un contexte de production de logiciel

- utilisation de types plus généraux ou plus proches des problèmes et détermination de stratégies de représentation vers les types de la base actuelle,
- adjonction de connaissances relatives au domaine de travail pour améliorer l'efficacité du système en prenant en compte des contraintes et propriétés particulières à un domaine,
- développement de bases de connaissances pour le choix de composants dans d'autres domaines que l'implantation de données abstraites.

Les études fondamentales à mener pour y parvenir touchent à l'analyse statique et dynamique d'algorithmes, à l'expression de stratégies de dérivation d'algorithmes [GRAM,86] qui peuvent contenir des informations pour guider les choix de représentation, à l'expression de critères de choix, à la complexité des programmes ; certaines sont voisines de celles dites de qualimétrie du logiciel [VER,86]. Elles devraient permettre de passer à un système expert dit de "seconde génération", doté de connaissances profondes sur les algorithmes (syntaxe, sémantique, complexité, qualimétrie) et n'interrogeant l'utilisateur que sur les points où sa collaboration est indispensable.

Les outils nécessaires sont des analyseurs d'algorithmes moins lourds que MENTOR, plus puissants, plus fins et facilement interfaçables avec d'autres applications, des langages moins rigides qu'ADA, favorisant l'écriture, la composition et la réutilisation de composants, des primitives d'expressions de stratégie en programmation.

Sous l'angle de l'enseignement de la programmation

Notre objectif était de proposer un environnement permettant à l'apprenant de se familiariser avec les raisonnements à mettre en oeuvre lorsqu'on fait des choix de représentations physiques d'objets dans le processus de construction d'un programme. Du point de vue de l'enseignement de l'informatique, nous apportons :

- un historique et un bilan critique de l'évolution des concepts, méthodes et outils en matière de représentation de données, un esposé des points de convergence des différents écrits dans ce domaine, notamment des principaux manuels d'enseignement et leur intégration dans un environnement d'enseignement dont nous avons conscience du caractère éphémère tant l'évolution est rapide (il faudra mettre entre les mains des étudiants des outils de transformation de types par exemple dès qu'ils seront disponibles, alors que notre projet n'aborde pas du tout cette question), mais dont nous affirmons l'intérêt comme étape dans ce processus inachevé de la formalisation des connaissances utilisées en représentation des données,
- un environnement qui ne s'adresse pas à des débutants en programmation, comme c'est le cas de la plupart des systèmes existants, mais qui vise au contraire à faciliter l'apprentissage de concepts avancés dans cette discipline (programmation abstraite, types de données, choix d'implantation efficace d'un objet, paquetage et généricité en ADA,...), qui permet de travailler dans un langage utilisé par les professionnels,

- un environnement qui n'est pas limité à des applications déterminées par l'enseignant mais dans lequel tout nouvel exercice peut être résolu,

- un environnement comprenant une bibliothèque de modules prédéfinis destinés à être réutilisés, évitant l'éternelle création de programmes ex-nihilo, permettant la lecture de textes bien structurés donnés aussi comme modèles, favorisant la programmation par modification de modules existants à coté de la création et de la réutilisation,

- une possibilité de confronter les principes d'une programmation descendante (méthode la plus souvent retenue dans un développement par niveaux d'abstraction successifs) avec les contraintes de la réutilisation de composants et les limites ainsi posées dans l'expression des algorithmes (on retrouve ainsi le contexte de travail du programmeur chaque fois qu'il modifie un environnement de programmation existant),

- une base de connaissances exprimant des critères de choix de représentation de données et permettant de suivre pas à pas des raisonnements conduits lors des choix, ce qui existe au mieux pour un ou deux exemples dans les manuels ou polycopiés usuels.

Nous avons indiqué comment enrichir la maquette réalisée du point de vue de l'assistance à la construction de programmes dans le paragraphe précédent, nous soulignons maintenant les aides qui pourraient être ajoutées du point de vue de l'enseignement des représentations de données pour l'élève et l'enseignant utilisateurs d'un tel système :

- une première aide serait un mode de structuration souple de l'univers des types et de celui des représentations, permettant de faire apparaître des structurations différentes suivant les besoins, par exemple de regrouper les listes et les arbres et de faire ressortir leurs traits communs avant de les différencier,

- pour l'enseignant qui veut personnaliser son environnement d'enseignement à partir du cadre proposé, une question essentielle est celle de l'ajout ou du retrait de types, d'opérations sur les types et de représentations pour un type, une autre est celle de l'expression déclarative des stratégies de raisonnement qu'il veut proposer à ses élèves ; enfin et avec une vision peut-être moins réaliste des difficultés, beaucoup rêvent à un système qu'on puisse paramétrer avec le langage d'expression d'algorithme que l'on a envie d'utiliser,

- à l'élève, il faudrait donner la possibilité de réponses "pourquoi pas" ou "pourquoi telle prémisse dans telle règle" et illustrer les réponses du système par des exemples et des contre-exemples ; on pourrait ensuite tenir compte de ces réponses de l'élève pour proposer d'autres exercices mettant en jeu les mêmes raisonnements ou des raisonnements voisins ;

- on pourrait également prendre en compte les réactions de l'élève au cours d'une session, analyser les réponses et essayer de passer d'une stratégie pédagogique à une autre (plus ou moins de réponses "je ne sais pas" autorisées, questions du système plus ou moins détaillées, etc...) selon les cas ;

- A coté du logiciel lui-même, il faudrait disposer d'une banque d'exercices, par exemple structurer les exercices des manuels en rapport avec les savoirs mis en évidence dans la base de connaissances, trouver de bons exercices illustrant bien la nécessité de raisonnements nuancés, la mise en oeuvre de stratégies de raisonnement différentes et la

pertinence de certaines règles ; cela permettrait d'aider les enseignants et d'envisager une utilisation en autoformation en plus de celle pilotée par le professeur préconisée aujourd'hui.

Enfin, une réflexion analogue pourrait être conduite pour la représentation des informations dans la construction de programmes dans des environnements différents de ceux des langages impératifs, en PROLOG ou SMALLTALK par exemple. De même, la réflexion pédagogique manque sur la coopération des approches ascendantes et descendantes dans la construction d'un programme avec un tel environnement, notamment des exemples pertinents de conflits entre les deux approches et de solutions pour les résoudre ou les éviter.

D'autres recherches fondamentales concerneraient l'analyse de l'activité de construction de programmes et des représentations que se font les professionnels par des psychologues et l'utilisation des observations recueillies dans un modèle de l'élève apprenti-programmeur.

Les outils qui ont fait défaut ne sont pas propres à l'enseignement de l'informatique, ils sont beaucoup plus généraux et nous les mentionnons au paragraphe suivant.

Construction de bases de connaissances et d'environnements à finalités pédagogiques

Dans ce domaine, nous avons essentiellement utilisé des concepts, des techniques et des outils existants pour réaliser la base de connaissances et le système d'aide au choix d'implantations, mais, ce projet a été générateur de beaucoup de questions qui amènent à autant de jalons vers la définition d'environnements adaptés à la construction de systèmes à finalités pédagogiques.

Notre apport concerne alors :

- la description complète de différentes étapes de la réalisation d'une première version de la base, notamment la mise en évidence et la structuration des concepts du domaine ; de telles descriptions dans des domaines variés nous semblent nécessaires pour la formation des enseignants à l'écriture et à l'utilisation de bases de connaissances dans la discipline qu'ils enseignent,

- la définition d'une "couche pédagogique" à partir du noyau initial,

illustrant une façon de séparer expertise disciplinaire et expertise pédagogique avec des outils conceptuels relativement simples, les règles de production,

- l'énoncé de conditions qui devraient caractériser le développement de l'utilisation de systèmes à bases de connaissances dans l'enseignement, notamment la mise en évidence des limites des outils actuels, mais le passage obligé par des outils de ce type pour entraîner rapidement les travaux d'expression d'expertise disciplinaire et pédagogique qui font défaut, même lorsqu'on dispose d'environnements sophistiqués.

Ce travail peut être prolongé dans les directions suivantes :

- repérage des connaissances de nature pédagogique qui peuvent influer sur la configuration d'un environnement d'enseignement par un professeur et expression de différentes stratégies induites par des données de cette nature (par exemple les acquis des élèves dans la discipline, les objectifs en matière de savoirs et de savoir-faire, les caractéristiques du public comme son aptitude à l'abstraction, sa vitesse dans l'acquisition de notions nouvelles),

- définition de primitives offertes à l'enseignant dans un langage admettant des métarègles et permettant d'engendrer automatiquement l'environnement adapté, dans le ces de SAIDA, les enchaînements d'étapes correspondants ; un ouvrage est difficilement paramétrable, c'est à dire adaptable aux besoins d'un individu donné, un logiciel devrait l'être davantage et ces propositions vont dans ce sens ;

- modification assistée ou automatique de la base de connaissances existante pour ajouter ou enlever un concept intermédiaire ou terminal,

- extraction de sous-bases cohérentes destinées à l'enseignement à partir de bases de connaissances du domaines construites à des fins d'utilisation professionnelles,

Les études fondamentales nécessaires concernent essentiellement l'expression de métaconnaissances, et notamment de métaconnaissances pédagogiques, le maintien de la cohérence dans les bases de connaissances, la conception et la structuration des bases selon différents points de vue, la recherche de critères permettant de proposer les formes de représentation de connaissances les mieux adaptées à un domaine donné.

Les outils nécessaires sont des générateurs de systèmes à base de connaissances, intégrant des types de connaissances variés et offrant toute une palette de représentations pour ces connaissances, plusieurs modes de raisonnement et facilitant l'expression d'expertise pédagogique.

Il faut également disposer de bases de connaissances et de formalisation des raisonnements dans lesquelles soit mise en évidence la notion d'étape; cette notion se révèle en effet importante dans un cadre didactique ; il faut faire apparaître les choix difficiles et exhiber des critères de choix ; il faut aussi inventer des situations pédagogiques où il y a des choix à faire. Décider, c'est, le plus souvent choisir. Apprendre aux élèves à décider, c'est les confronter à de multiples situations dans lesquelles ils ont des choix à faire, et donc recenser ou inventer de telles situations.

Il manque enfin des facilités d'intégration de logiciels et de communication d'informations, notamment une normalisation des représentations de bases de connaissances si l'on veut facilement utiliser des bases construites par d'autres.

La tendance est actuellement à l'émergence de générateurs de systèmes experts adaptés à des familles d'applications (diagnostic, planification, temps réel, etc...), ce travail fournit des jalons pour la définition d'environnements de création de systèmes pédagogiques à bases de connaissances, on pourrait parler de générateurs de tuteurs intelligents.

Notre apport, sous l'angle de l'ordinateur outil d'enseignement, comprend :

- une maquette d'environnement d'enseignement à la fois ouvert à l'élève (pas d'exercices pré-enregistrés, accès à d'autres fonctions et d'autres logiciels à partir de l'application) et ouvert à l'enseignant (configuration de versions d'environnement par catégorie d'étudiants, paramétrisation possible des contenus, et, à contenu déterminé, des modes de raisonnement),

- l'existence dans cet environnement d'un système à base de connaissances qui n'en n'est pas nécessairement le centre, mais est utilisé à la demande et à pour objectif d'entraîner aux raisonnements nécessaires pour les choix d'implantation de données, et ce faisant, de montrer les abstractions intermédiaires utilisées pour conduire ces raisonnements,

- l'identification d'acteurs divers dans les processus de construction, modification et utilisation du système, ainsi que des fonctions remplies par chacun d'eux :

Acteurs en phase de conception :
L'**auteur** de l'application qui définit les objectifs, choisit l'architecture générale et pilote l'ensemble de la réalisation.
L'**expert du domaine** apporte la connaissance et permet sa formalisation, avec l'aide éventuelle d'un **cogniticien**.
Le pédagogue du domaine, il est expert du domaine, mais plus encore de l'enseignement dans ce domaine ; il connait les difficultés d'apprentissage des élèves, sait décrire et utiliser des stratégies pédagogiques différentes selon les traits de son public ; il réalise la partie pédagogique de la base de connaissances.
Le technicien (dans SAIDA, le programmeur par exemple) peut être chargé de la réalisation d'auxiliaires nécessaires (la bibliothèque de procédues ADA).

Acteurs en phase d'utilisation :
Le **responsable d'enseignement** qui configure le système selon les besoins des différents groupes d'apprenants, son rôle est en fait d'extraire un sous-système de l'environnement proposé et de l'initialiser (droits, noms d'élèves, etc...).
L'**enseignant** qui n'intervient pas sur la configuration mais suit le travail de ses élèves.
Le **groupe d'élèves** qui partage des droits et ressources propres, définis à partir de profils pédagogiques communs et des objectifs de l'enseignant.
L'**élève** que le système connait par son nom, sa catégorie, éventuellement d'autres informations concourant à la définition d'un modèle de celui-ci.
L'ingénieur-système qui gére les ressources communes comme les mots de passe, l'espace disque, etc...

On pourrait évidemment identifier d'autres acteurs pour d'autres systèmes, comme un **psychopédagogue** pour des stratégies relevant de la pédagogie générale, ou un chercheur en **psychologie cognitive** ou encore un **spécialiste multimédia** pour des parties sonores ou graphiques.

En termes d'outils, cela signifie que, dans un système de conception, il faut offrir des fonctionnalités différentes à chacun de ces acteurs selon leurs caractéristiques et leurs besoins propres.

Si "chercher, c'est apporter des réponses à quelques questions, mais aussi découvrir et formuler des interrogations nouvelles", nous pensons avoir fait oeuvre de recherche. Notre souhait, au terme de cette étude, est de pouvoir nous engager avec d'autres dans quelques-unes des multiples pistes que nous avons contribué à tracer.

Annexes

Exemple de programme

Bases de règles

Programme du Hit-Parade

```
with text_io;
with LIST_MANAGER; -- package pour definir les operations sur liste
use text_io;

procedure HIT_PARADE is

 -- types necessaires ------------------------------------------------------
subtype t_choix is integer range 0..100; -- references chansons disponibles
subtype t_cptr is integer range 0..1000; -- nombre de personnes interrogees
type tab_choix is array(1 .. 5) of t_choix;
subtype tnom is string(1..30);
type t_enr is   record  nom : tnom;
                        sexe : integer range 1..2 ; -- 1/masculin 2/feminin
                        age : integer range 0..99;
                        l_choix : tab_choix;  -- de t_choix (5)
                end record;
type t_tube is record   num: t_choix;
                        score : t_cptr;
                end record;

 -- declarations generiques ------------------------------------------------
package mon_entier is new integer_io(integer); use mon_entier;
package mes_donnees is new list_manager(elem => t_enr); use mes_donnees;
package mes_intermediaires is new list_manager(elem => t_cptr);
 use mes_intermediaires;
package mes_tubes is new list_manager(elem => t_tube); use mes_tubes;
package mes_aux is new list_manager(elem => tnom); use mes_aux;

 -- donnees ----------------------------------------------------------------
l_donnees : mes_donnees.liste; -- de t_enr
l_resultat : mes_tubes.liste;
        -- de t_tubes N elements ordonnes par val decroiss de score
l_intermediaire: mes_intermediaires.liste;      -- de t_cptr objet intermediaire
l_f_inf20: mes_aux.liste;       -- de nom des femmes age <20
l_f_sup20: mes_aux.liste;       -- de nom des femmes age >=20
l_h_inf20: mes_aux.liste;       -- de nom des hommes age <20
l_h_sup20: mes_aux.liste;       -- de nom des hommes age >=20

nb_chansons : t_choix; -- nombre total de chansons

 -- procedures annexes-----------------------------------------------------

 -- initialisation de la liste de donnees
Procedure INIT_L_DONNEE(l_donnee : in out mes_donnees.liste) is
        e: t_enr;
        encore : boolean:= true;

        procedure SAISIR_UN(e : in out t_enr ; encore :in out boolean) is
        rep : character;
        begin
        put("Donner le nom");get(e.nom);
        put("Donner le sexe 1=homme 2= femme");get(e.sexe);
        put("Donner l age"); get(e.age);
        put("Donner les cinq numeros de chansons dans l ordre");
        for i in 1..5 loop
                get(e.l_choix(i));
        end loop;
        put("On continue ? o/n ");get(rep);
        encore := (rep = 'o');
        end SAISIR_UN;
        procedure SAISIR_TOUS is new mes_donnees.INIT_LIST(saisie => saisir_un);

        begin
        saisir_tous(l_donnee);
end INIT_L_DONNEE;

 -- initialisation de la liste intermediaire
Procedure INIT_L_INTER(l_inter : in out mes_intermediaires.liste) is

        nb :t_choix;
        e: t_cptr;
        encore : boolean := true;

        procedure SAISIR_UN(e : in out t_cptr ; encore :in out boolean) is
        begin
        nb := nb+1; -- modification du nombre de chansons saisies
        e := 0;
        encore := (nb = nb_chansons);
        end SAISIR_UN;
        procedure SAISIR_TOUS
                is new mes_intermediaires.INIT_LIST(saisie => saisir_un);
```

```
        begin
        nb := 0;
        saisir_tous(l_inter);
end INIT_L_INTER;

 -- modification de la liste intermediaire l_inter
 -- parcours total de la liste de donnees l_data sans modifications
Procedure REMPLIR_LISTE_INTER ( l_donnee : in out mes_donnees.liste;
                        l_inter : in out mes_intermediaires.liste) is

        derniere_place : mes_donnees.place;
        procedure COMPTABILISE_UN (enr: in out t_enr) is
         -- incremente 1 a 5 compteurs pour chaque chanson citee
        compte :t_cptr;
        num_chanson : t_choix;
        begin
          for i in 1 .. 5 loop
          num_chanson :=enr.l_choix(i);
          compte:=mes_intermediaires.valeur(L_INTER,acces(l_inter,num_chanson))+1;
          mes_intermediaires.chglp(L_INTER,num_chanson,compte);
        end loop;
        end COMPTABILISE_UN;
        procedure COMPTABILISE_TOUS
                is new mes_donnees.PARC_TOTAL(fonc =>comptabilise_un);

        begin
        comptabilise_tous(L_DONNEE, derniere_place);
end REMPLIR_LISTE_INTER;

 -- creation de la liste resultat par tri de la liste intermediaire
 -- adjonction d'un tube bien place dans la liste resultat
Procedure REMPLIR_LISTE_RES (l_inter : in out mes_intermediaires.liste ;
                        l_res : in out mes_tubes.liste) is

        -- recherche du premier emplacement de l_res
        -- tel que le score est inferieur
        -- a celui de la nouvelle chanson a inserer dans le hit
        -- variable globale utilisee w_compte
        derniere_place : mes_intermediaires.place;
        num_chanson : t_choix :=1; -- le rang represente le numero de chanson
        val : t_tube;

        function PLUS_PETIT(e: in t_tube) return boolean is
        begin
        return( e.score <val.score);
        end plus_petit;
        procedure PREMIER_PLUS_PETIT
                is new mes_tubes.PREM_TEL_QUE(predicat => plus_petit);

        procedure TRIE_UN (compte : in out t_cptr) is
        rg : mes_tubes.trang;
        pl : mes_tubes.place;
        begin
        -- le nouvel element a ranger dans l_res
        val.num := num_chanson;
        val.score := compte;
        -- cherche dans la liste l_res le premier score inferieur a celui donne
        premier_plus_petit(L_RES,rg,pl);
        if rg = 0
                then mes_tubes.adjl(L_RES,num_chanson,val);
                else mes_tubes.adjl(L_RES, rg-1,val);
        end if;
        num_chanson := num_chanson + 1;
        end TRIE_UN;

        procedure TRIE_TOUS is new mes_intermediaires.PARC_TOTAL(fonc =>trie_un);

        begin
        num_chanson := 1;
        -- parcours et place chaque tube correspondant a l_inter
        trie_tous(L_INTER, derniere_place);
end REMPLIR_LISTE_RES;
```



```
        -- creation des autres listes resultat avec parcours de l_data
        -- variables globales utilisees top1,top2,top3

procedure REMPLIR_AUX (L_RES : in out mes_tubes.liste;
        L_DATA : in out mes_donnees.liste;
        L_H_INF20, L_H_SUP20, L_F_INF20, L_F_SUP20 : in out mes_aux.liste) is

        derniere_place : mes_donnees.place;
        top1,top2,top3 : t_choix;

        procedure TESTER_UN (e : in out t_enr) is
        nb_choix : integer range 0..5;
        num_chanson :t_choix;
        arret : boolean := false;
        begin
        nb_choix := 0;
        while not arret loop
          nb_choix := nb_choix +1;
          num_chanson:=e.l_choix(nb_choix);
          if num_chanson = top1 or num_chanson = top2 or num_chanson = top3
          then arret := true;
               if e.sexe = 1

               then if e.age <20
                    then mes_aux.adjl(L_H_INF20,mes_aux.queue(L_H_INF20),e.nom);
                    else mes_aux.adjl(L_H_SUP20,mes_aux.queue(L_H_SUP20),e.nom);
                    end if;
               else if e.age <20
                    then mes_aux.adjl(L_F_INF20,mes_aux.queue(L_F_INF20),e.nom);
                    else mes_aux.adjl(L_F_SUP20,mes_aux.queue(L_F_SUP20),e.nom);
                    end if;
               end if;
           else arret := (nb_choix >=5);
           end if;
        end loop;
        end TESTER_UN;
        procedure TESTER_TOUS is new mes_donnees.PARC_TOTAL(fonc=>TESTER_UN);

        begin
        top1:= mes_tubes.valeur(L_RES,acces(L_RES,1)).num;
        top2:= mes_tubes.valeur(L_RES,acces(L_RES,2)).num;
        top3:= mes_tubes.valeur(L_RES,acces(L_RES,3)).num;
        tester_tous(L_DATA,derniere_place);
end REMPLIR_AUX;

 -- programme principal ------------------------------------------------
        begin
        get(nb_chansons);
        mes_donnees.creer_list(l_donnees);
        init_l_donnee(l_donnees);

        mes_intermediaires.creer_list(l_intermediaire);
        init_l_inter(l_intermediaire);
        remplir_liste_inter(l_donnees, l_intermediaire);

        mes_tubes.creer_list (l_resultat);
        remplir_liste_res (l_intermediaire, l_resultat );

        mes_aux.creer_list(l_f_inf20);
        mes_aux.creer_list(l_f_sup20);
        mes_aux.creer_list(l_h_inf20);
        mes_aux.creer_list(l_h_sup20);

        remplir_aux(l_resultat, l_donnees,
                        l_h_inf20, l_h_sup20, l_f_inf20, l_f_sup20);

end HIT_PARADE;
```

Base de règles relatives aux listes

```
-------------------------------------------------------------------------------
        Base de regles pour conclure ou pencher en faveur d'une conclusion
-------------------------------------------------------------------------------

Si la liste existe deja , l'implantation est imposee
 alors implantation = implantation imposee .

        -- pour conclure sur le support ---
Si non des problemes de place en memoire centrale ,
 non obligation d'utiliser un fichier
 alors support = interne .

Si obligation d'utiliser un fichier
 alors support = externe .

Si la place memoire est suffisante ,
 non obligation d'utiliser un fichier
 alors support = interne .

Si non la place memoire est suffisante
 alors support = externe .

        -- pour choisir le fichier --
Si support = externe , il y a des modifications ,
 non il y a des suppressions , non il y a des adjonctions
 alors implantation = fichier a acces direct .

Si non il y a des adjonctions  , non il y a des modifications ,
 non il y a des suppressions , support = externe
 alors implantation = fichier sequentiel .

Si support = externe , il y a des suppressions
 alors implantation = fichier a acces direct avec reserves .

Si support = externe , il y a des adjonctions
 alors implantation = fichier a acces direct avec reserves .

        -- pour estimer les problemes d'encombrement memoire --
Si la place en memoire = grande ou assez grande ,
 la taille de la liste = petite ou moyenne
 alors la place memoire est suffisante .

Si la place en memoire = petite ou limitee ,
 la taille de la liste = assez grande ou grande ou tres grande
 alors non la place memoire est suffisante .

        -- pour a conclure sur le nombre d'elements de la liste --
Si il n'y a que des suppressions ,
 non les suppressions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements =  variable .

Si il n'y a que des suppressions ,
 les suppressions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements =  varie tete/queue .

Si il n'y a que des adjonctions ,
 non les adjonctions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements =  variable .

Si il n'y a que des adjonctions ,
```

```
Si il y a beaucoup d'adjonctions ,
 non les adjonctions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements = varie beaucoup .

Si il y a beaucoup d'adjonctions ,
 les adjonctions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements = varie tete/queue .

Si il y a beaucoup de suppressions ,
 non les suppressions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements = varie beaucoup .

Si il y a beaucoup de suppressions ,
 les suppressions ont lieu exclusivement en tete (en queue)
 alors les variations du nombre d'elements = varie tete/queue .

        -- pour conclure sur la representation --
Si la liste est entierement creee en sequence , il n'y a que des modifications ,
 non les acces se font sequentiellement
 alors representation = contigue .

Si la taille de la liste est bornee , la taille de la liste = petite ou moyenne
 alors representation = contigue .

Si la taille de la liste = petite  ou moyenne ou grande ,
 les variations du nombre d'elements = fixe ou peu variable ou varie tete/queue
 alors representation = contigue .

Si beaucoup plus d'acces que d'adjonction/suppression ,
 les variations du nombre d'elements = peu variable ou fixe ou globalement fixe
 alors representation = contigue .

Si il y a des adjonctions ,
 non beaucoup plus d'acces que d'adjonction/suppression ,
 les adjonctions ont lieu exclusivement en tete (en queue)
 alors representation2 = contigue .

Si il y a des suppressions ,
 non beaucoup plus d'acces que d'adjonction/suppression ,
 les suppressions ont lieu exclusivement en tete (en queue)
 alors representation2 = contigue .

Si les variations du nombre d'elements = peu variable  ou variable
 ou globalement fixe ou varie tete/queue
 alors representation2 = contigue .

Si le rang a une signification semantique dans l'application ,
 la majorite des acces se fait par reference au rang ,
 non les acces se font sequentiellement
 alors representation2 = contigue .

Si une partie des acces se fait par valeur
 alors representation2 = contigue .

Si il y a beaucoup de modifications
 alors representation2 = contigue .

Si les variations du nombre d'elements =  variable
 alors representation1 = chainee .

Si les variations du nombre d'elements = varie beaucoup
 alors representation1 = chainee .

Si il y a des adjonctions , il y a des suppressions ,
```



```
Si il y a des adjonctions , il y a des suppressions ,
 le nombre des adjonctions compense a peu pres le nombre des suppressions ,
 les adjonctions ne se placent au meme rang que les suppressions
 alors representation1 = chainee .

Si il y a des adjonctions ,
 non beaucoup plus d'acces que d'adjonction/suppression ,
 non les adjonctions ont lieu exclusivement en tete (en queue) ,
 non  la taille de la liste est bornee
 alors representation = chainee .

Si il y a des suppressions ,
 non beaucoup plus d'acces que d'adjonction/suppression ,
 non les suppressions ont lieu exclusivement en tete (en queue) ,
 non  la taille de la liste est bornee
 alors representation = chainee .

        -- pour parcourir la liste dans les deux sens --
Si on utilise souvent la notion de predecesseur (de precedant)
 alors chainage = double .

Si l'ordre des parcours est inverse a l'ordre de creation
 alors chainage = double .

        -- pour organiser les valeurs --
Si on peut etablir une relation d'ordre sur la valeur des elements ,
 la relation d'ordre sur valeur est utilisee dans les parcours ,
 on s'autorise a toucher a l'ordre des elements
 alors organisation = tri .

Si on peut etablir une relation d'ordre sur la valeur des elements ,
 la relation d'ordre doit apparaitre dans l'organisation de la liste ,
 on s'autorise a toucher a l'ordre des elements
 alors organisation = tri .

Si on peut etablir une relation d'ordre sur la valeur des elements ,
 non  la relation d'ordre doit apparaitre dans l'organisation de la liste ,
 non on s'autorise a toucher a l'ordre des elements ,
 organisation a conserver = triee
 alors organisation = tri .

Si on peut etablir une relation d'ordre sur la valeur des elements ,
 non  la relation d'ordre doit apparaitre dans l'organisation de la liste ,
 non on s'autorise a toucher a l'ordre des elements ,
 organisation a conserver = quelconque
 alors organisation = quelconque .

Si il y a beaucoup de recherche par valeur ,
 on peut etablir une relation d'ordre sur la valeur des elements ,
 la taille de la liste est bornee ,
 on s'autorise a toucher a l'ordre des elements
 alors organisation = tri .

        ---- Traitement des conclusions ---
Si support = interne ,
 representation = chainee ,
 chainage = simple ,
 organisation = quelconque
 alors implantation = liste chainee simple ordre quelconque .

Si support = interne ,
 representation = chainee ,
```

```
 alors implantation = liste chainee simple triee .

Si support = interne ,
 representation = chainee ,
 chainage = double ,
 organisation = quelconque
 alors implantation = liste chainee double ordre quelconque .

Si support = interne ,
 representation = chainee ,
 chainage = double ,
 organisation = tri
 alors implantation = liste chainee double triee .

Si support = interne ,
 representation = contigue ,
 organisation = quelconque
 alors implantation = tableau contigu  ordre quelconque .

Si support = interne ,
 representation = contigue ,
 organisation = tri
 alors implantation = tableau contigu trie .

 ------------------------------------------------------------------------
        Raffinement de la base et traitement des deductions intermediaires
        elles representent des orientations de choix
 ------------------------------------------------------------------------


        ----  Adjonction des regles valeur par defaut ----

Si representation ?? alors representation = contigue .

Si chainage ?? alors chainage = simple .

Si support  ?? alors support = interne .

Si organisation  ?? alors organisation = quelconque .

Si on peut utiliser un objet intermediaire en memoire centrale
 alors support = interne .

Si on s'autorise a toucher a l'ordre des elements ??
 alors on s'autorise a toucher a l'ordre des elements .

        ---- Traitement des conclusions partielles ----
Si representation1 = chainee , representation2 = contigue ,
 beaucoup plus d'acces que d'adjonction/suppression ,
 on souhaite privilegier les acces ,
 on souhaite optimiser la gestion de la place memoire ??
 alors privilegier contigue .

Si representation1 = chainee , representation2 = contigue ,
 beaucoup plus d'acces que d'adjonction/suppression ,
 on souhaite privilegier les acces ,
 non on souhaite optimiser la gestion de la place memoire
 alors privilegier contigue .

Si representation1 = chainee , representation2 = contigue ,
 privilegier contigue
 alors representation = contigue .

Si representation1 ?? , representation2 = contigue
```



```
Si representation1 = chainee , representation2 = contigue ,
 privilegier contigue ??
 alors representation = chainee .

Si on souhaite optimiser la gestion de la place memoire ,
 il existe plusieurs listes de taille variable ,
 representation2 = contigue
 alors representation = chainee .

Si representation1 = chainee , representation2 ??
 alors representation = chainee .

 ------------------------------------------------------------------------
              ---- Des regles pour eviter les questions parasites ----
 ------------------------------------------------------------------------
  ---- Regles de traductions noms de procedures operations sur les objets ----
Si val alors il y a des acces .
Si supl alors il y a des suppressions .
Si adjl alors il y a des adjonctions .
Si chglp alors il y a des modifications .
Si parc-partiel alors il y a des acces .
Si parc-total alors il y a des acces , les acces suivent l'ordre de creation .
Si premier-tel-que alors il y a des acces .

Si non la liste existe deja alors non la liste existe dans un fichier .

Si non des problemes de place en memoire centrale
 alors la place memoire est suffisante .

Si non adjl alors non il y a des adjonctions .
Si non il y a des adjonctions alors non il y a beaucoup d'adjonctions .
Si non il y a des adjonctions alors non il n'y a que des adjonctions .
Si non il y a des adjonctions
 alors non les adjonctions ont lieu exclusivement en tete (en queue) .
Si non il y a des adjonctions alors non adjl .
Si non adjl alors non il n'y a que des adjonctions .

Si non il y a des modifications alors non il y a beaucoup de modifications .
Si non il y a des modifications alors non il n'y a que des modifications .
Si non chglp alors non il y a des modifications .

Si non supl alors non il y a des suppressions .
Si non il y a des suppressions alors non il y a beaucoup de suppressions .
Si non il y a des suppressions alors non supl .
Si non il y a des suppressions
 alors non les suppressions ont lieu exclusivement en tete (en queue) .
Si non supl alors non il n'y a que des suppressions .

Si non adjl , non supl
 alors les variations du nombre d'elements = fixe ,
       beaucoup plus d'acces que d'adjonction/suppression .

Si la majorite des acces se fait par valeur
 alors non la majorite des acces se fait par reference au rang .
Si la majorite des acces se fait par reference au rang
 alors non la majorite des acces se fait par valeur .

        ---- Regles traduisant des relations de type implication ---
Si la majorite des acces se fait par valeur
 alors une partie des acces se fait par valeur .
Si non une partie des acces se fait par valeur
 alors non la majorite des acces se fait par valeur .
```

```
        Regles pour exprimer differents niveaux de modelisation de l'univers
        exemple : estimation de l'encombrement memoire
 ------------------------------------------------------------------------
        -- pour selectionner des pre-requis suivant niveau d'enseignement --
Si et/peda3
 alors $choix-attributs-faut-savoir-experimente  .

        -- pour aider a conclure sur la taille de la liste --
Si le type de l'element = une structure a champs fixes
 alors estimation de la taille de la liste = possible .

Si le type de l'element = un  tableau  de dimensions connues
 alors estimation de la taille de la liste = possible .

Si le type de l'element = une structure avec un champ variable ou
 un  tableau  de dimensions inconnues
 alors estimation de la taille de la liste = impossible .

Si les elements sont de type simple
  ,  la taille de la liste est bornee
 alors estimation de la taille de la liste = possible .

Si le type de l'element = complexe
 alors estimation de la taille de la liste = impossible .

Si estimation de la taille de la liste ??
 alors la taille de la liste = assez grande .

Si la taille d'un element = petite ,
 le nombre d'elements = petit ou moyen
 alors la taille de la liste = petite .
Si la taille d'un element = moyenne ,
 le nombre d'elements = petit ou moyen
 alors la taille de la liste = moyenne .
Si la taille d'un element = grande ,
 le nombre d'elements = grand ou moyen
 alors la taille de la liste = grande .

         -- pour aider a conclure sur la taille d'un element --
Si le type de l'element est un type simple predefini dans le langage
 alors la taille d'un element =  petite .

Si le type de l'element = entier ou booleen ou reel ou caractere
 alors les elements sont de type simple , la taille d'un element =  petite .

Si le type de l'element = une chaine de caracteres
 alors  la taille d'un element = moyenne .


 ------------------------------------------------------------------------
        Regles qui aident l'utilisateur non experimente
 ------------------------------------------------------------------------
        -- pour completer l'algorithme --
Si la liste a une valeur initiale connue
  ,  non la liste existe deja
 alors adjonction debut algo  = il faut initialiser la liste .

Si obligation d'utiliser un fichier
 alors adjonction debut algo = ouvrir le fichier ,
 adjonction fin algo = fermer le fichier .

Si la duree de la liste est superieure  a celle du programme
 alors on pour utiliser un objet intermediaire en memoire centrale ,
```



```
Si on doit minimiser le temps d'execution ,
 la liste existe deja , la liste existe dans un fichier ,
 la place memoire est suffisante
 alors adjonction debut algo = vider le fichier ,
        adjonction fin algo = recopier dans un fichier ,
        support = interne .

Si  support = externe
 alors adjonction  debut algo = ouvrir le fichier
  ,  adjonction  fin algo = fermer le fichier .

        --  des connaissances sur les acces --
Si la majorite des acces se fait par valeur
 alors remise en question de la liste .

 ------------------------------------------------------------------------
        Structuration du raisonnement : Enchainement des etapes
 ------------------------------------------------------------------------
        -- pour definir l'enchainement selectionne dans pedal --
Si choix = cop alors et/contraintes .
Si choix = cop , fin-sortie-et/contraintes alors et/objets .
Si choix = cop , fin-sortie-objets alors  et/procedures .

Si choix = cpo alors et/contraintes .
Si choix = cpo , fin-sortie-et/contraintes alors et/procedures .
Si choix = cpo , fin-sortie-et/procedures alors et/objets .

Si choix = opc alors et/objets .
Si choix = opc , fin-sortie-et/objets alors et/procedures .
Si choix = opc , fin-sortie-et/procedures alors et/contraintes .

Si choix = ocp alors et/objets .
Si choix = ocp , fin-sortie-et/objets alors et/contraintes .
Si choix = ocp , fin-sortie-et/contraintes alors et/procedures .

Si choix = poc alors et/procedures .
Si choix = poc , fin-sortie-et/procedures alors et/objets .
Si choix = poc , fin-sortie-et/objets alors et/contraintes .

Si choix = pco alors et/procedures .
Si choix = pco , fin-sortie-et/procedures alors et/contraintes .
Si choix = pco , fin-sortie-et/contraintes alors et/objets .

  -- la valeur est erronnee ou inconnue on affecte une valeur par defaut --
Si choix ?? alors choix = cop .

Si support ??  alors $envisager ( et/contraintes ) .

Si des problemes de place en memoire centrale ??
alors et/taille/liste , $envisager ( et/mixage ) .

        -- pour gerer les enchainements des etapes --
Si des problemes de place en memoire centrale
 alors $envisager ( et/mixage ) .

Si support = interne alors $envisager ( et/chaine ) .
Si support = interne alors $envisager ( et/tri ) .
Si support = interne alors et/fichier = faux .

Si support = externe alors $envisager ( et/fichier ) .
Si support = externe alors $envisager ( et/tri ) .
Si suport = externe alors et/chaine = faux .
```

```
Si le type de l'element est un type simple predefini dans le langage ??
alors $envisager ( et/type ) .

Si non le type de l'element est un type simple predefini dans le langage
alors $envisager ( et/type ) .

Si implantation ?? alors $envisager ( et/fin ) .
Si implantation ?? , fin-sortie-et/fin alors $envisager ( et/fin/partielle ) .

Si implantation * alors et/fin = vrai .

Si fin-sortie-et/peda2 alors  et/fin = vrai .
                -- pour se substituer a cond-arret --
        -- l'etape doit etre activee avant d'etre arretee --

Si non des problemes de place en memoire centrale
 alors et/disponible = faux .

Si estimation de la taille de la liste = impossible
 alors et/taille/liste = faux .

Si la taille d'un element * alors et/taille/elem = faux .

Si non il y a des adjonctions  alors et/adjonction = faux .
Si non il y a des adjonctions  alors et/adjonc/sup = faux .

Si non il y a des adjonctions alors et/adjonc/sup = faux .

Si il n'y a que des adjonctions  alors et/suppression = faux .
Si il n'y a que des adjonctions  alors et/adjonc/sup = faux .

Si non il y a des suppressions  alors et/suppression = faux .
Si non il y a des suppressions  alors et/adjonc/sup = faux .

Si il n'y a que des suppressions alors et/adjonc/sup = faux .
Si il n'y a que des suppressions alors et/adjonction = faux .

Si non il y a des modifications alors et/modification = faux .

Si non on peut etablir une relation d'ordre sur la valeur des elements
 alors et/acces = faux .
Si non on peut etablir une relation d'ordre sur la valeur des elements
 alors et/parcours = faux .

Si support = externe alors et/chaine = faux .

Si representation = contigue alors et/simple = faux .

Si support = interne alors et/fichier = faux .
```


Base de règles relative aux ensembles

```
-------------------------------------------------------------------------------
                    implantation du type ensemble
        base de regles pour conclure ou pencher en faveur d'une conclusion
-------------------------------------------------------------------------------
        -- parce qu'on ne peut choisir --

Si l'ensemble etudie est le resultat d'une recopie
 alors
 conseil1 = prendre en compte la representation de l'ensemble copie .

        -- parce qu'on peut tenir compte de ce qui existe --
Si l'ensemble etudie est le resultat d'une operation ensembliste
 alors conseil1 = prendre en compte les representations des antecedants .


   -- la meilleure representation: valider l'intermediaire vers-booleens --

Si les elements du referentiel sont connus pour chaque execution ,
 le referentiel de l'ensemble est partage avec d'autres ensembles
alors vers-booleens .

Si les elements du referentiel sont connus pour chaque execution ,
 taille ensemble est en permanence bien plus petite que celle du referentiel ,
 operations de parcours = exclusivement
alors non vers-booleens .

Si non les elements du referentiel sont connus pour chaque execution
alors non vers-booleens .

Si les elements du referentiel sont connus pour chaque execution ,
 element peut servir d'index dans un tableau en ADA ,
 la taille de l'ensemble est voisine de celle du referentiel
alors vers-booleens .

Si les elements du referentiel sont connus pour chaque execution ,
 non le referentiel de l'ensemble est partage avec d'autres ensembles ,
 la taille de l'ensemble est voisine de celle du referentiel ,
 non element peut servir d'index dans un tableau en ADA
alors non vers-booleens .

 ---------- on se dirige vers l'intermediaire tableau de valeurs ------------
Si le type de l'element = entier ou caractere ou enumere ou
 chaine de caracteres ou structure a champs fixes
 alors elements homogenes .

Si le type de l'element = complexe ou structure avec un champ variable
 alors non elements homogenes .

Si il existe une borne connue de la taille de l'ensemble ,
 les variations du nombre d'elements = fixe ,
 elements homogenes
alors tableau de valeurs des elements = souhaite .

Si il existe une borne connue de la taille de l'ensemble ,
 vers-booleens ,
 non le referentiel de l'ensemble est partage avec d'autres ensembles ,
 on souhaite favoriser les operations acces/recherche
alors tableau de valeurs des elements = souhaite .

Si il existe une borne connue de la taille de l'ensemble ,
 les variations du nombre d'elements = varie ou varie beaucoup ,
 elements homogenes
alors tableau de valeurs des elements = possible .
```

```
alors tableau de valeurs des elements = refuse .

Si non elements homogenes
alors tableau de valeurs des elements = refuse .

 ------------------------- organisation triee ou non ----------------------
Si non il existe un ordre sur les elements
alors organisation = ordre quelconque .

Si il existe un ordre sur les elements ,
 on souhaite favoriser les test d'app/min/max
alors organisation = triee .

Si il existe un ordre sur les elements ,
 il existe des operations ensemblistes avec l'objet comme composant
alors organisation = triee .

 ------------ on cherche a conclure sur l'intermediaire vers-chainee ---------
Si les variations du nombre d'elements = varie ou varie beaucoup
alors vers-chainee .

Si non elements homogenes
alors vers-chainee .

Si non il existe une borne connue de la taille de l'ensemble
alors vers-chainee .

Si on souhaite gerer au mieux l'espace memoire ,
 il y a beaucoup d'adjonctions ,
 non la taille de l'ensemble est voisine de celle du referentiel
alors vers-chainee .

Si beaucoup d'adjonctions/suppressions en position quelconque
alors vers-chainee .

Si vers-chainee ,
 arbre binaire de recherche = refuse ,
 liste chainee avec hachage = refuse
alors liste chainee de valeurs = souhaite .

---------------------- On exprime des conclusions finales ---------------------
                --- sur tableau de booleens --

Si vers-booleens ,
 element peut servir d'index dans un tableau en ADA ,
 la taille de l'ensemble est voisine de celle du referentiel
alors tableau de booleens = souhaite .

Si vers-booleens ,
 non la taille de l'ensemble est voisine de celle du referentiel ,
 il existe des operations ensemblistes avec l'objet comme composant
alors tableau de booleens = souhaite .

Si vers-booleens ,
 non la taille de l'ensemble est voisine de celle du referentiel ,
 operations de parcours = majoritairement
alors tableau de booleens = possible .

                --- sur les tableaux de valeurs ---
Si organisation = triee ,
 tableau de valeurs des elements = souhaite
alors  tableau trie de valeurs des elements = souhaite .

Si organisation = ordre quelconque ,
 tableau de valeurs des elements = souhaite
```



```
alors  tableau ordre quelconque de valeurs des elements = souhaite .

Si organisation = triee ,
 tableau de valeurs des elements = possible
alors  tableau trie de valeurs des elements = possible .

Si organisation = ordre quelconque ,
 tableau de valeurs des elements = possible
alors  tableau ordre quelconque de valeurs des elements = possible .

Si tableau de valeurs des elements = refuse
alors tableau ordre quelconque de valeurs des elements = refuse ,
tableau trie de valeurs des elements = refuse .

                --- sur les arbres binaires ---
Si organisation = ordre quelconque
alors arbre binaire de recherche = refuse .

Si il y a necessite d'optimiser les temps d'acces ,
 il y a beaucoup d'adjonctions ,
 les variations du nombre d'elements = varie ou varie beaucoup ,
 il y a plus d'adj/supp que de consultations seules ??
alors arbre binaire de recherche = possible .

Si il y a necessite d'optimiser les temps d'acces ,
 il y a beaucoup d'adjonctions ,
 les variations du nombre d'elements = varie ou varie beaucoup ,
 non il y a plus d'adj/supp que de consultations seules
alors arbre binaire de recherche = souhaite .

Si  les variations du nombre d'elements = fixe
alors arbre binaire de recherche = refuse .

Si des problemes de place en memoire centrale
alors arbre binaire de recherche = refuse .

Si vers-chainee ,
 des problemes de place en memoire centrale ,
alors arbre binaire de recherche = refuse .

Si vers-chainee ,
 on souhaite favoriser les operations de recherche ,
 non il existe une bonne fonction de hachage  ,
 non des problemes de place en memoire centrale
alors arbre binaire de recherche = souhaite .

                --- sur les listes chainees ---
Si il y a necessite d'optimiser les temps d'acces ,
 il y a beaucoup d'adjonctions ,
 les variations du nombre d'elements = varie ou varie beaucoup ,
 il existe une bonne fonction de hachage ,
 il y a plus d'adj/supp que de consultations seules
alors   liste chainee avec hachage = souhaite .

Si il y a necessite d'optimiser les temps d'acces ,
 il y a beaucoup d'adjonctions ,
 les variations du nombre d'elements = varie ou varie beaucoup ,
 il existe une bonne fonction de hachage ,
 il y a plus d'adj/supp que de consultations seules ??
alors liste chainee avec hachage = possible .

Si vers-chainee ,
 on souhaite favoriser les operations acces/recherche ,
 il existe une bonne fonction de hachage
alors   liste chainee avec hachage = souhaite .
```

```
Si non il existe une bonne fonction de hachage
alors   liste chainee avec hachage = refuse .

Si il existe un ordre sur les elements ,
 liste chainee de valeurs = souhaite ,
 on souhaite favoriser les test d'app/min/max
alors liste chainee de valeurs triee = souhaite .

Si il existe un ordre sur les elements ,
 liste chainee de valeurs = choisie par defaut ,
 on souhaite favoriser les test d'app/min/max
alors liste chainee de valeurs triee = choisie par defaut .

Si organisation = triee ,
 liste chainee de valeurs = souhaite ,
alors liste chainee de valeurs triee = souhaite .

Si organisation = ordre quelconque ,
 liste chainee de valeurs = souhaite ,
alors liste chainee de valeurs dans un ordre quelconque = souhaite .

                --- sur les fichiers a acces direct ---
Si le support est impose a fichier acces direct
alors fichier a acces direct = souhaite .

Si la taille de l'ensemble est trop importante pour etre traitee en memoire centr
alors fichier a acces direct = souhaite .

Si non le support est impose a fichier acces direct ,
non la taille de l'ensemble est trop importante pour etre traitee en memoire cent
alors fichier a acces direct = refuse .

---------------- pour arreter la recherche quand on a conclu ----------------
Si tab de bool indexe par le type de l'element de l'ensemble = souhaite
alors arret .
Si tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem =
souhaite alors arret .
Si tableau trie de valeurs des elements =  souhaite alors arret .
Si tableau ordre quelconque de valeurs des elements = souhaite alors arret .
Si arbre binaire de recherche = souhaite alors arret .
Si liste chainee avec hachage = souhaite alors arret .
Si liste chainee de valeurs triee = souhaite alors arret .
Si liste chainee de valeurs dans un ordre quelconque  = souhaite alors arret .
Si fichier a acces direct = souhaite alors arret .
Si arbre binaire de recherche = possible ,
 liste chainee avec hachage = possible
alors arret .
-------------------------------------------------------------------------------
        Raffinement de la base et traitement des deductions intermediaires
-------------------------------------------------------------------------------
                ---- Pour affiner le resultat ----
Si tableau de booleens = souhaite ,
 element peut servir d'index dans un tableau en ADA
alors tab de bool indexe par le type de l'element de l'ensemble = souhaite .

Si tableau de booleens = souhaite ,
 non element peut servir d'index dans un tableau en ADA
alors
tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem = souhait
conseil2 = il faut initialiser le referentiel dans l'algorithme .

Si tableau de booleens = possible ,
 element peut servir d'index dans un tableau en ADA
alors tab de bool indexe par le type de l'element de l'ensemble = possible .
```



```
Si tableau de booleens = possible ,
 non element peut servir d'index dans un tableau en ADA
alors
tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem = possible ,
conseil2 = il faut initialiser le referentiel dans l'algorithme .

        ---- En cas de dualite entre conclusions regles de priorite ----

        -- pour differencier liste chainee et table de valeurs --
Si tableau de valeurs des elements = possible ,
 vers-chainee ,
 on souhaite favoriser les operations acces/recherche
alors tableau de valeurs des elements = souhaite .

Si tableau de valeurs des elements = possible ,
 vers-chainee ,
 on souhaite favoriser les operations d'adj/supp
alors liste chainee de valeurs = souhaite .

        ---- Adjonction de regles valeur par defaut ----

Si il existe une borne connue de la taille de l'ensemble ,
les variations du nombre d'elements ?? ,
 elements homogenes
alors tableau de valeurs des elements = possible .

Si organisation ??
alors organisation = ordre quelconque .

                -- pour conclure quand meme --

Si liste chainee de valeurs ?? , arret ??
alors liste chainee de valeurs = choisie par defaut .

Si organisation = ordre quelconque ,
 liste chainee de valeurs = souhaite ,
alors liste chainee de valeurs dans un ordre quelconque = souhaite , arret .

Si organisation = ordre quelconque ,
 liste chainee de valeurs = choisie par defaut
alors liste chainee de valeurs dans un ordre quelconque = choisie par defaut .

Si organisation = ordre quelconque ,
 liste chainee de valeurs ?? ,
 arret ??
alors liste chainee de valeurs dans un ordre quelconque = choisie par defaut .

Si organisation = triee ,
 liste chainee de valeurs = choisie par defaut
alors liste chainee de valeurs triee = choisie par defaut .

Si organisation = triee ,
 liste chainee de valeurs ?? ,
 arret ??
alors liste chainee de valeurs triee = choisie par defaut .

Si organisation = ordre quelconque ,
 liste chainee de valeurs ?? ,
 arret ??
alors liste chainee de valeurs dans un ordre quelconque = choisie par defaut .

-------------------------------------------------------------------------------
        Des regles pour eviter des questions parasites
-------------------------------------------------------------------------------
```

```
Si operations de parcours  = exclusivement
alors non il existe des operations ensemblistes avec l'objet comme composant .

 -- une regle pour avoir une valeur dattribut supplementaire --
Si operations de parcours = neant alors $vide .

Si beaucoup d'adjonctions/suppressions en position quelconque
alors les variations du nombre d'elements = varie .

Si il y a beaucoup d'adjonctions
alors les variations du nombre d'elements = varie beaucoup .

Si la taille de l'ensemble est voisine de celle du referentiel
alors
non taille ensemble est en permanence bien plus petite que celle du referentiel

Si taille ensemble est en permanence bien plus petite que celle du referentiel
alors non la taille de l'ensemble est voisine de celle du referentiel .

Si la taille de l'ensemble est voisine de celle du referentiel
alors il existe une borne connue de la taille de l'ensemble .

        -- car le conflit est seulement entre op de modif/op d'acces --
Si on souhaite favoriser les operations acces/recherche
alors non on souhaite favoriser les operations d'adj/supp .

Si on souhaite favoriser les operations d'adj/supp
alors non on souhaite favoriser les operations acces/recherche .

                -- des deductions internes --
Si non vers-booleens
alors tableau de booleens = refuse .

Si tableau de booleens = refuse
alors tab de bool indexe par le type de l'element de l'ensemble = refuse ,
 tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem = refu

Si le type de l'element = entier ou caractere ou enumere
alors  element peut servir d'index dans un tableau en ADA .

Si le type de l'element = chaine de caracteres ou complexe
 ou structure avec un champ variable ou structure a champs fixes
alors  non element peut servir d'index dans un tableau en ADA .
```

Structuration du raisonnement : Enchainement des etapes

```
 ---- on procede par raffinements successifs ----
Si fin-sortie-et/fichier  alors $envisager ( et/tab/bool ) .

Si fin-sortie-et/tab/bool alors $envisager ( et/tab/val ) .

Si fin-sortie-et/tab/val alors $envisager ( et/chaine ) .

 ---- on abandonne des orientations devenues caduques ---

Si non les elements du referentiel sont connus pour chaque execution
alors  $envisager ( et/tab/val ) .

Si non il existe une borne connue de la taille de l'ensemble
 alors $envisager ( et/chaine ) .

Si non il existe un ordre sur les elements alors et/arbre = faux .
```



```
Si le support est impose a fichier acces direct alors ss/et/fichier = faux .

 ---- on oriente le raisonnement dans le cas d'abandon ---

        -- pour substituer les conditions d'arret des etapes --
Si arret alors et/chemin = faux .
Si arret alors et/tab/bool  = faux .
Si arret alors et/tab/val = faux .
Si tableau de valeurs des elements = refuse alors et/tab/val = faux .
Si arret alors et/chaine  = faux .
Si arret alors et/vite = faux .
Si arret alors et/tout = faux .
```

```
;-------------------------------------------------------------------------
;       une etape de depart
;-------------------------------------------------------------------------
ETAPE et/depart :
ENTREE
!ETAPE et/depart :
! ---------------------------------------------------------------------
! Reservee a l'enseignant en phase de test du systeme expert
! Pour choisir une strategie de travail
! --> pour QUITTER : entrez "$"
! ---------------------------------------------------------------------
! --> pour selectionner une ETAPE c'est a dire une methode :
! entrez le nom de l'etape que vous souhaitez activer puis entrez "fin"
! .....................................................................
! la liste des etapes possibles est :
! et/chemin : on suit la constitution de la base
! et/vite : on cherche a conclure le plus rapidement possible
! et/tout : on examine tous les attributs recenses
! ---------------------------------------------------------------------
$insertion-faits .
OBLIGATOIRE
et/fin .
;-------------------------------------------------------------------------
;       une etape destinee a ne presenter que les conclusions interessantes
;-------------------------------------------------------------------------
ETAPE et/fin :
SORTIE
!ETAPE et/fin : voici les conclusions
conseil1 ,
tab de bool indexe par le type de l'element de l'ensemble ,
tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem ,
conseil2 ,
tableau trie de valeurs des elements ,
tableau ordre quelconque de valeurs des elements ,
liste chainee avec hachage ,
arbre binaire de recherche ,
liste chainee de valeurs triee ,
liste chainee de valeurs dans un ordre quelconque ,
fichier a acces direct ,
! Des explications
! Pour demander des explications sur un attribut donnez son nom
! pour terminer tapez ---> fin
! pour quitter tapez  ---> $
$insertion-sorties .
;-------------------------------------------------------------------------
;       une demarche pedagogique possible
;-------------------------------------------------------------------------
ETAPE et/chemin :
ENTREE
!ETAPE et/chemin :
! On suit la demarche qui est celle de la mise en place de la base
.
OBLIGATOIRE
et/issu/de , et/fichier .
ENVISAGEABLE
et/tab/bool , et/tab/val , et/hachage , et/arbre , et/chaine .
COND-ARRET
arret .
;-------------------------------------------------------------------------
ETAPE et/issu/de :
ENTREE
!ETAPE et/issu/de : Si l'objet etudie est issu d'objets existant
!on donne des informations complementaires
.
prmc
```



```
conseil1 .
;-------------------------------------------------------------------------
ETAPE et/tab/bool :
ENTREE
!ETAPE et/tab/bool : Pour conclure sur tableau de booleens
le type de l'element ,
les elements du referentiel sont connus pour chaque execution .
BUTS
vers-booleens * ,
tab de bool indexe par le type de l'element de l'ensemble ,
tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem .
COND-ARRET
arret .
;-------------------------------------------------------------------------
ETAPE et/tab/val :
ENTREE
!ETAPE et/tab/val : pour conclure sur un tableau de valeurs
le type de l'element ,
elements homogenes ,
il existe une borne connue de la taille de l'ensemble .
BUTS
tableau de valeurs des elements , organisation .
COND-ARRET
tableau de valeurs des elements = refuse .
;-------------------------------------------------------------------------
ETAPE et/chaine :
ENTREE
!ETAPE et/chaine : pour conclure sur une representation chainee
des problemes de place en memoire centrale .
OBLIGATOIRE
et/arbre , et/hachage .
BUTS
liste chainee de valeurs triee ,
liste chainee de valeurs dans un ordre quelconque .
COND-ARRET
arret .
;-------------------------------------------------------------------------
ETAPE et/hachage :
ENTREE
!ETAPE et/hachage : pour conclure sur le hachage
il existe une bonne fonction de hachage .
BUTS
liste chainee avec hachage .
;-------------------------------------------------------------------------
ETAPE et/arbre :
ENTREE
!ETAPE et/arbre : pour conclure sur un arbre binaire de recherche
il existe un ordre sur les elements .
OBLIGATOIRE
ss/et/arbre .
BUTS
arbre binaire de recherche .
;.....................
ETAPE ss/et/arbre :
ENTREE
!ETAPE ss/et/arbre : pour justifier un choix couteux en place memoire
les variations du nombre d'elements ,
il y a necessite d'optimiser les temps d'acces ,
des problemes de place en memoire centrale .
COND-ARRET
non il existe un ordre sur les elements .
;-------------------------------------------------------------------------
ETAPE et/fichier :
ENTREE
```

```
le support est impose a fichier acces direct .
OBLIGATOIRE
ss/et/fichier .
BUTS
fichier a acces direct .
;.....................
ETAPE ss/et/fichier :
ENTREE
!ETAPE ss/et/fichier :
la taille de l'ensemble est trop importante pour etre traitee en memoire centrale
COND-ARRET
le support est impose a fichier acces direct .
;-----------------------------------------------------------------
;       une autre demarche possible qui organise les saisies
;-----------------------------------------------------------------
ETAPE et/vite :
ENTREE
!ETAPE et/vite : pour conclure au plus vite on elague
le type de l'element ,
les elements du referentiel sont connus pour chaque execution ,
il existe un ordre sur les elements ,
les variations du nombre d'elements .
BUTS
tableau de booleens ,
tab de bool indexe par le type de l'element de l'ensemble ,
tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem ,
tableau de valeurs des elements ,
tableau trie de valeurs des elements ,
tableau ordre quelconque de valeurs des elements ,
liste chainee de valeurs ,
organisation ,
liste chainee avec hachage ,
arbre binaire de recherche ,
liste chainee de valeurs triee ,
liste chainee de valeurs dans un ordre quelconque ,
fichier a acces direct .
COND-ARRET
arret .
;-----------------------------------------------------------------
;       une autre demarche possible pour envisager tous les problemes
;-----------------------------------------------------------------
ETAPE et/tout :
ENTREE
!ETAPE et/tout :
! on etudie successivement les contraintes , les objets, les procedures
.
OBLIGATOIRE
et/contraintes , et/objets , et/procedures .
BUTS
conseil1 ,
tab de bool indexe par le type de l'element de l'ensemble ,
tab de bool indexe par un interv d'ent avec rech preal du rg de l'elem ,
tableau trie de valeurs des elements ,
tableau ordre quelconque de valeurs des elements ,
liste chainee avec hachage ,
arbre binaire de recherche ,
liste chainee de valeurs triee ,
liste chainee de valeurs dans un ordre quelconque ,
fichier a acces direct .
COND-ARRET
arret .
;-----------------------------------------------------------------
ETAPE et/procedures :
ENTREE
!ETAPE et/procedures : on etudie les operations
```



```
il existe des operations ensemblistes avec l'objet comme composant ,
operations de parcours ,
il y a beaucoup d'adjonctions ,
beaucoup d'adjonctions/suppressions en position quelconque ,
on souhaite favoriser les test d'app/min/max ,
on souhaite favoriser les operations acces/recherche ,
on souhaite favoriser les operations d'adj/supp
.
;-----------------------------------------------------------------
ETAPE et/objets :
ENTREE
!ETAPE et/objets : on etudie les objets
le type de l'element ,
les variations du nombre d'elements ,
les elements du referentiel sont connus pour chaque execution ,
il existe un ordre sur les elements ,
il existe une bonne fonction de hachage ,
l'ensemble etudie est le resultat d'une recopie ,
l'ensemble etudie est le resultat d'une operation ensembliste ,
il existe une borne connue de la taille de l'ensemble ,
la taille de l'ensemble est voisine de celle du referentiel ,
taille ensemble est en permanence bien plus petite que celle du referentiel ,
la taille de l'ensemble est trop importante pour etre traitee en memoire centrale
.
;-----------------------------------------------------------------
ETAPE et/contraintes :
ENTREE
!ETAPE et/contraintes : on etudie les contraintes
le support est impose a fichier acces direct ,
on souhaite gerer au mieux l'espace memoire ,
des problemes de place en memoire centrale ,
il y a necessite d'optimiser les temps d'acces
.
;-----------------------------------------------------------------
FIN
```

Bibliographie

Bibliographie

[ABB,87] ABBOT R.J.
Knowledge abstraction
CACM, vol.30, n°8, pp.664-671, 1987

[ABR,78] ABRIAL J.R.
Z: A Specification Language
IFIP, Tokyo, 1978

[ADA,80] ADAM A., LAURENT J.P.
LAURA : A system to debug student programs.
Artificial Intelligence, vol.15, pp.75-122, 1980

[ADE,82] CHEVAL J.L., ESTUBLIER J., GHOUL S., KRAKOWIAK S.
Modularité et composition des programmes dans l'atelier de logiciels
Adèle, Actes du 1° Colloque AFCET de Génie Logiciel, Paris 1982

[AHO,74] AHO A., HOPCROFT J., ULLMAN J.
The Design and Analysis of Computer Algorithms
Addison Wesley, Reading, Mass., 1974

[AHO,83] AHO A., HOPCROFT J., ULLMAN J.
Data Structures and Algorithms
Addison Wesley, Reading, Mass., 1983

[ALE,88] ALEXANDRE F., FINANCE J.P., QUERE A.
SPES: un système de transformation de programmes logiques
Colloque de Programmation Logique, Trégastel, 1988

[AND,84] ANDERSON J.R., BOYLE C.F., FARRELL R.G., REISER B.J.
Cognitive Principles in the Design of Computer Tutors
Proc. 6th Annual Conf. of the Cognitive Sci. Soc., Boulder co, June 84

[AND,85] ANDERSON J.R., REISER B.J.
The LISP Tutor, BYTE, vol.10, n°4, pp.159-175, 1985

[AND,86] ANDERSON J.R., SKWARECKI E.
The Automated Tutoring of Introductory Computer Programming
CACM, vol.39, n°9, 1986.

[ARS,77] ARSAC J.
Nouvelles Leçons de Programmation
Dunod, Paris, 1977

[ARS,83] ARSAC J.
Les bases de la programmation
Dunod, Paris, 1983

[ARS,87] ARSAC J.
Garder le cap, Revue Informatiques, n°1, CRDP de Poitiers, 1987

[AYE,86] AYEL M.,PIPARD E.,ROUSSET M.C.
Le contrôle de cohérence dans les bases de connaissances.
Journées sur l'Intelligence artificielle, PRC-GRECO, Aix-les-Bains, Novembre 1986, Cepadues Editions, Toulouse. 1986

[BAL,73] BALZER R.
A global view of automatic programming
3rd. IJCAI, pp. 494-499, 1973

[BAL,85] BALZER R.
A 15-year perspective on automatic programming
IEEE Transactions on Software Engineering, SE-11, 11, 1985

[BAR-ON,87] BAR-ON E., OR-BACH R.
Explanation based learning in intelligent tutoring systems, IFIP TC3, Frascati, LEWIS R., ERCOLI P. (eds.), North Holland, 1987

[BARD,85] BARDIN B.M.
A "to be determined" Package for ADA development.
ACM Ada Letters, Nov. 1985

[BARD,87] BARDIN B.M., THOMPSON J.
Composable Ada Software Components and the Re-Export Paradigm
ACM Ada LETTERS, vol.8, n°1, 1988

[BARN,84] BARNES J.
Programming in ADA
Addison Wesley, London, 1984

[BARON,84] BARON M.
D'un SE à un SEIAO, les enseignements d'un cas intéressant GUIDON, notes de lecture. Colloque Intelligence Artificielle, Aix-en-Pce, publication n°49 du GR22, CNRS, 1984

[BARON,87] BARON M.,VIVET M.
Systèmes experts et tuteurs intelligents
Congrès AFCET RFIA, Antibes, 1987

[BARR,81] BARR A., FEIGENBAUM E. (eds.)
The Handbook of Artificial Intelligence
Heuristec Press, Stanford, Calif. 1981

[BARS,82] BARSTOW D.R.
The roles of knowledge and deduction in algorithm design
Machine Intelligence n°10, Addison Wesley, 1982

[BARS,79a] BARSTOW D.R.
An experiment in knowledge-based automatic programming
Artificial Intelligence, vol.12, pp.73-119, 1979

[BARS,79b] BARSTOW D.R.
Knowledge-based program construction, Elsevier, North Holland, 1979

[BARS,84a] BARSTOW D.R.
A perspective in automatic programming
AI Magazine, pp.5-27, Spring 1984

[BARS,85a] BARSTOW D.R.
Domain specific automatic programming
IEEE Transactions on Software Engineering, SE-11, 11, 1985

[BARS,85b] BARSTOW D.R.
On convergence toward a data base of program transformations
ACM Toplas, vol.7, n°1, pp.1-9, 1985

[BARS,84b] BARSTOW D.R., SHROBE H.E., SANDEWALL E. (eds.)
Interactive programming environments
Mac Graw Hill, New York, 1984

[BART,81] BARTELS U., OLTHOFF W, RAULEFS P.
APE: An expert system for automatic programming from abstract specification of data types and algorithms. Memo SEKI BN-81-01, Institut fur Informatik, Bonn /Karlsruhe 1981

[BART,82] BARTELS U.,OLTHOFF W.,RAULEFS P.
An expert system for implementing abstract sorting algorithms on parametrized abstract data types. 7th. IJCAI, Vancouver, Canada, 1982.

[BAU,82] BAUER F.L.
From specifications to machine-code: program construction through formal reasoning. 6th. Internat. Conference on Software Engineering,1982.

[BENCH,86] BENCHIMOL et al.
Systèmes experts dans l'entreprise.
Hermès, Paris 1986.

[BENCI,79] BENCI G., ROLLAND C.
Les bases de données: D'une conception canonique à une réalisation extensible SCM, Paris, 1979

[BENTL,79] BENTLEY J.L., SHAW M.
An ALPHARD Specification of a correct and efficient Transformation of data structures Proc. of Specifications of Reliable Sofware,
IEEE Catalog 79-CH-14019C, 1979

[BER,79] BERT D.
La programmation générique: Construction de logiciel, Spécification algébrique et Vérification Thèse d'Etat,
Université Scient. et Méd. de Grenoble, 1979

[BES,82] BESTOUGEFF H., FARGETTE J.P.
Enseignement et Ordinateur
Cedic / Nathan, Paris, 1982

[BID,81] BIDOIT M.
Une méthode de présentation de types abstraits : Applications.
Thèse de 3° cycle, Université de Paris Sud Orsay, LRI, 1981

[BID,85] BIDOIT M., GRESSE C., LOSAVIO F.
Apex: Un système de construction assistée de programmes Ada fondé sur la prise en compte de cas d'exceptions, Journées GROPLAN, Toulouse, 1985.
Les langages et leurs environnements, BIGRE+GLOBULE n°46., 1985

[BID,87] BIDOIT M.,CAPY F.,CHOPPY C.,CHOQUET N.,GRESSE C.,KAPLAN S.,SCHLIENGER F.,VOISIN F.
ASSPRO: un environnement de programmation interactif et intégré.
TSI, vol.6, n°1, pp.21-39, 1987

[BID,86] BIDOIT M.,CHOPPY C., VOISIN F.
The ASSPEGIQUE specification environment : Motivations and design.
Rapport LRI,1986

[BIE,84] BIERMANN A.,GUIHO G.,KODRATOFF Y. (eds.)
Automatic program construction techniques
Mac Millan pub. comp., New York, 1984

[BIO,83] BIONDI, CLAVEL
Introduction à la programmation
Masson , Paris, 1983

[BIR,77] BIRTWISTLE G.H.,DAHL O.Y.,MIRHAUG B.,NIGAARD K.
SIMULA Begin
Petrocelli / Charter, New York, 1977

[BOL,88] BOLOGNA S.
Software measurement ? What and How ? An application in the field of safety for powerplants
Actes du 1° séminaire européen sur la qualité des logiciels, AFCIQ, Bruxelles,1988

[BON,84] BONNET A.
L'intelligence artificielle, promesses et réalités.
Inter-éditions, Paris 1984.

[BON,86] BONNET A., HATON J.P., TRUONG-NGOC J.M.
Sytèmes Experts: Vers la maitrise technique, Interéditions, 1986

[BOO,83] BOOCH G.
Software Engineering with ADA
Benjamin/Cummings Pub. Comp.,1983

[BOO,87] BOOCH G.
Software Components with ADA: Structures, Tools and Subsystems
Benjamin/Cummings Pub. Comp., 1987

[BOUR,86] BOURGUIGNON J.P.
La structure d'accueil Emeraude
Revue Génie Logiciel n°6, Nov. 1986

[BOUS,77] BOUSSARD J.C., LECARME O.
Didactique de la Programmation
Ecole d'été de l'AFCET, Montreal, 1977

[BOUS,84] BOUSSARD J.C., MAHL R.
Programmation avancée
Eyrolles, Paris, 2° édition, 1984.

[BRAN,86] BRAND D.
On typing in Prolog
ACM Sigplan notices, vol.21, n°1, pp.28-30, 1986

[BRAS,86] BRASSARD G., BRATLEY P.
Algorithmique: conception et analyse.
Masson, 1986

[BRE,87] BREUKER J., WINKELS R., SANDBERG J.
A shell for intelligent help systems
IJCAI 87

[BROD,84] BRODIE M.J., MYLOPOULOS J., SCHMIDT J. (eds.)
On Conceptual Modeling: Perspectives from Artificial Intelligence, Data Bases and Programming Languages Springer Verlag, New York, 1984

[BROU,87] BROUAYE P., BRUILLARD E., FERRET E., WEIDENFELD G.
APPAT: Un tuteur intelligent pour l'apprentissage des tableurs par la résolution de problèmes. Congrès francophone EAO 87, Cap d'Agde, pp.247-256. 1987

[BUL,88] BULL
Introduction à KOOL (document interne Cie Bull), 1988

[BUR,77] BURSTALL R., DARLINGTON J.
A transformation system for developing recursive programs
JACM, vol.24, n°1, 1977

[CARBO,70] CARBONEL J.R.
AI in CAI: an artificial intelligence approach to computer assisted instruction
IEEE Trans. on man-mach. systems., vol MMS 11, 1970

[CARDE,85] CARDELLI L., WEGNER P.
On Understanding Types, Data Abstractions and Polymorphism.
Computing Surveys, vol.17, n°4, 1985

[CHER,75] CHERBONNEAU B.,GALINIER M.,LAGASSE J.P.,MASSIE H.,MATHIS B.,PAUL J.L.
Programmation Structurée
Ecole d'été de l'AFCET, Rabat, 1975

[CHO,87a] CHOURAQUI E., INGHILTERRA C.
Apport de la méthodologie fondée sur les objets pour la construction d'un système d'EAO de la géométrie. Congrès MARI-COGNITIVA 87, pp.39-44 1987

[CHO,87b] CHOURAQUI E., INGUILTERRA C.
Conception d'une base de connaissances orientée objet pour l'EAO de la géométrie Colloque francophone EAO 87, Cap d'Agde, pp.61-70, 1987

[CIP,81] Groupe CIP
Report on a wide spectrum language for program specification and development, Springer Verlag, LNCS-183, 1981

[CLAE,88] CLAES G. et al.
Starguide: un générateur de systèmes d'autoformation à l'usage de logiciels TSI, vol.7,n°1, 1988

[CLAN,82] CLANCEY W.J.
GUIDON in The Handbook of Artificial Intelligence, pp.267-278 BARR, FEIGENBAUM (eds.), William Kaufmann, 1982

[CLAN,87] CLANCEY W.J.
Qualitative models and instruction: an overview of GUIDON2 research 3rd. internat. conf. on AI and Education, Pittsburgh, 1987

[CLAN,83] CLANCEY W.J. The epistemology of rule based expert systems: a framework for explana- tion Artificial Intelligence, vol.20, pp.215-250, 1983

[CLAN,81] CLANCEY W.J., LETSINGER R.
NEOMYCIN: Reconfiguring a rule-based expert system for application to teaching, 7th. IJCAI, pp.828-836, 1981

[CLE,87] CLEO (Centre Lorrain d'Enseignement assisté par Ordinateur)
Projets retenus après l'appel d'offres de l'ADI "systèmes experts et enseignement", Frouard, 1987.

[COO,86] COOK M.E., du BOULAY B.
A pascal program checker
ECAI 86, pp.90-95, 1986

[COR,84] CORDIER M.O., FALLER B., KAYSER D., NICAUD J.F.
EIAO : Une application des systèmes experts à l'EAO. Bulletin de l'association EPI, Décembre 1984, pp.15-44.

[COU,87a] COULETTE B.
IPHIGENIE, un didacticiel expert en méthodologie de développement de projets logiciels, Congrès francophone EAO 87, Cap d'Agde, pp.207-218, 1987

[COU,87b] COULETTE B.
IPHIGENIE: Un système expert pour la formation au génie logiciel
Journées Systèmes Experts, Avignon, 1987

[COUR,87] COURTIN J., KOWARSKI I.
Initiation à l'algorithmique et aux structures de données
Dunod, 1987

[COUV,84] COUVERT A., PEDRONO R.
Techniques de programmation, Polycopié, cours C45. Université de Rennes 1, Laboratoire d'Informatique, 1984

[DAH,72] DAHL O.J., DIJKSTRA E.W., HOARE C.A.R.
Structured Programming
Academic Press, London, 1972

[DAR,81] DARLINGTON J.
An experimental program transformation and synthesis system, Artificial Intelligence, vol.16, pp.1-46, 1981

[DAT,75] DATE C.J.
Introduction to Data Base Systems
Addison Wesley, 1975

[DAV,82] DAVIS R., LENAT D.B.
Knowledge-based Systems in AI.
Mac Graw Hill, New York, 1982

[DAV,77] DAVIS R.,BUCHANAN S.,SHORTLIFFE E.
Production Rules as a Representation for a Knowledge-Based Consultation Program. Artificial Intelligence, vol. 8, n°1, Feb. 1977.

[DE MAS,85] DE MASSAS E., LOTT M.
Un langage informatique universel ADA
La Recherche, n°164, pp.349-358, 1985

[DED,86] DEDE C.
A review and synthesis of recent research in intelligent computer assisted instruction., Int. Journ.Man-Mach. Studies, vol.24, pp.329-353, 1986.

[DEL,87] DELAYHE J.P.
Systèmes Experts : organisation et programmation de bases,
Eyrolles, Paris, 1987

[DER,79] DERNIAME J.C., FINANCE J.P.
Types abstraits de données ; spécification, réalisation, utilisation.
Ecole informatique de l'AFCET, Monastir, Tunisie, 1979

[DEW,82] DEWAR R.B.K., SCHONBERG E., SCHWARZ J.T.
Higher level programming : An introduction to the Programming Language SETL, Courant Institute of math. studies, New York University, 1982

[di SESSA,85] di SESSA A.A.
Social niches for future software
3rd. int. conf. on AI and Education, Pittsburgh, May 1985

[DIJ,76] DIJKSTRA E.W.
A Discipline of programming
Prentice Hall, Englewood Cliffs,N.J.,1976

[DONA,85] DONAHUE J., DEMERS A.
Data Types are Values
ACM Toplas, 7, vol.3, pp.426-445, 1985

[DONZ,80] DONZEAU-GOUGE V.,HUET G.,KAHN G.,LANG B.
Programming Environments Based on Stuctured Editors: The Mentor Experience Rapport INRIA n°26, Juillet 1980.

[du BOU,87a] du BOULAY B.
What should a programming environment for novice programmers be like?
Third Int. Conf. on AI and Education, Pittsburgh, May 1987

[du BOU,87b] du BOULAY B.
Intelligent systems for teaching programming IFIP TC3, Frascati, May 1987 LEWIS R., ERCOLI P.(eds.), North Holland, 1987

[du BOU,85] du BOULAY B., ELSOM-COOK M., KHABASA T., TAYLOR J.
POPLOG and the learner : An artificial programming environment used in education Cognitive Studies Programme, (draft paper), University of Sussex, 1985

[du BOU,86] du BOULAY B., SOTHCOTT C.
Computers teaching programming: an introductory survey on the field in artificial intelligence and education, LAWLER R.W.,YASDANI M.(eds.) 1986

[DUC,84] DUCRIN A.
Programmation Tomes 1 et 2 Dunod, Paris, 1984

[EAR,71] EARLEY J.
Towards an understanding of data structures
CACM, vol.14, n°10, pp.617-627, 1971

[EDF,88] Electricité De France
L'intelligence artificielle, une réalité industrielle
8° Journées Journées Internationales les systèmes experts et leurs applications , Avignon, 1988, Catalogue de l'exposition, pp.136-137

[EMB,87] EMBLEY D.W., WOODFIELD S.N.
A knowledge structure for reusing abstract data types
9th. Software Engineering Conference, Monterey, 1987

[ERN,85] ERNST C.
Introduction aux systèmes experts de gestion.
Eyrolles, Paris 1985.

[ERN,69] ERNST G. NEWELL A.
GPS : A case study in generality
New York, Acad. Press, 1969

[EUR,87] EURATEC (Sté.)
CATALYST : apprendre à programmer en langage ADA.
Euratec, Paris, 1987.

[FAR,85] FARRENY H.
Les systèmes experts : principes et exemples.
Cepadues Editions, Toulouse, 1985.

[FEA,82] FEATHER M.S.
A system for Assisting Program Transformation
ACM Toplas, vol.4, n°1, pp.1-20, 1982

[FEI,82] FEIGENBAUM E.A. et al.
The Handbook of Artificial Intelligence.
Vol.1,2,3, W.Kaufmann, 1981/1982.

[FEL,85] FELDMAN M.B
Programming and Data Structure, Relting Publishing Company INC., 1985

[FER,88] FERNANDEZ I., DIAZ A., VERDEJO M.F.
Building a programming tutor by dynamic planning: case studies and a proposal Intelligent Tutoring Systems, Montreal, juin 1988

[FIN,78] FINANCE J.P.
De la spécification abstraite d'une donnée à sa représentation en mémoire: les états successifs d'une information, rapport CRIN 78-P-022, 1978

[FIN,79] FINANCE J.P.
Etude de la construction des programmes: méthodes et langages de construction et de résolution de problèmes, thèse d'Etat, Université Nancy1, 1979

[FIN,79] FINANCE J.P.,HUC B.,LESCANNE P.,PAIR C.,QUERE M.,REMY J.L.
Programmation déductive et structure de données
Rapport CRIN 79-E-051, 1979

[FIN,84] FINANCE J.P, GRANDBASTIEN M.,LEVY N.,QUERE A., SOUQUIERES J. : SPES: Un système pour vérifier et transformer.
Colloque AFCET, Génie Logiciel, Nice, 1984

[FIN,85] FINANCE J.P.
Cours de licence d'informatique, Université Nancy 1, non publié, 1985

[FIS,87] FISCHER G.
A critic for LISP, Proceedings IJCAI 87, pp.177-184, 1987

[FIS,85] FISCHER G.
From interactive to intelligent systems, in: the Challenge of advanced computing technology to system design methods Proc. of NATO Advanced Study Institute, Springer Verlag, Heidelberg, 1985

[FLA,80] FLAJOLET P., FRANCON J., VUILLEMIN J.
Sequence of operations analysis for dynamic data structures.
Journal of algorithms, 1, pp.11-141, 1980

[FLO,67] FLOYD R.W.
"Assigning meaning to programs" pp.10-32, Proc. Amer. Mat.Soc. Symp. in Applied Math., vol.19, Providence, Rhodes Island., 1967.

[FOI,85] FOISSEAU J.,JACQUARD R.,LEMAITRE M.,LEMOINE M.,ZANON G
Le système SPRAC: Expression et Gestion de Spécifications d'Algorithmes et de Représentation., TSI, vol.4, n°2, pp.237-254, 1985

[FOR,87] FORCHERI P., MOLFINO M.T
A historical approach to educational computer-based systems
Revue d'Intelligence Artificielle, vol.1, pp.53-70, Hermès, Paris,1987

[FOU,82] FOUCAUT O.
Modèle et outil pour la conception des systèmes d'informations dans les organisations, Thèse d'Etat, Université Nancy 1, 1982.

[FRA,88] FRANCON J., RANDRIANARIMANANA B., SCHOTT R.
Analysis of dynamic algorithms in D.E. Knuth's model
Proc. CAAP'88, Lecture Notes in Computer Science n°229, pp.72-88
Springer Verlag, 1988

[GAL,88] GALMICHE D.
Constructive system for automatic program systhesis
INFORMATIKA 88, Nice, Fev. 1988

[GAR,86] GARIJO F.,VERDEJO M.F.,HERNANDEZ L., INCAUSTI-IKERLAN
SINTALAB: An expert system for the synthesis of abstract algorithms 6 th. intern. workshop on expert systems and their applications Avignon, 1986

[GAU,78] GAUDEL M.C., PAIR C.
Structures de données et Algorithmes Fondamentaux, Cours INRIA, 1978

[GAU,86] GAUDEL M.C., SORIA M., FROIDEVAUX C
Types de données et algorithmes, vol. 1 et 2, Collection Didactique,
INRIA, Rocquencourt, 1986

[GAU,78] GAUDEL M.C., TERRINE G.
Synthèse de la représentation d'un type abstrait par des types concrets
Congrès AFCET panorama de l'Informatique, Ed. Hommes et Techniques,
Tome 1, Paris, 1978.

[GAUTH,86] GAUTHIER M.
La généricité et les listes
Bigre+Globule, Actes des Journées ADA AFCET / ENST, 1986

[GER,77] GERBIER (nom collectif)
Mes premières constructions de programmes
Springer Verlag, 1977

[GER,81] GERHART S.L.
Affirm reference manual
UCS Report, Marina del Rey, 1981

[GIB,84] GIBSON J.
POPLOG in New Horizons in Educational Computing, M.YASDANI (ed.)
Ellis Horwood, 1984

[GOG,78] GOGUEN J.J., THATCHER J., WAGNER E.
An initial approach to the specification, correctness and implementation of abstract data types in Current Trends in Programming Methodology, vol.4,
YEH ed.,Prent.Hall, 1978

[GOG,83] GOGUEN J.J.
Parametrized Programming
Proc. Workshop on reusability in programming, Stratford CT,USA, 1983

[GOG,86] GOGUEN J.J.
Reusing and interconnecting software components
IEEE Computer, vol.19, n°2, pp.16-28, 1986

[GOL,75] GOLDSTEIN J.P.
Summary of MYCROFT : A system for understanding simple picture programs, Artificial Intelligence, vol.6, 1975

[GRAM,86] GRAM A.
Raisonner pour programmer
Dunod, Paris, 1986

[GRAN,84] GRANDBASTIEN M.
Le développement de l'intelligence artificielle concerne tous les enseignants
Revue de l'association EPI, numéro spécial "Systèmes experts, déc.1984

[GRAN,86] GRANDBASTIEN M.
Evolution de l'informatique: du traitement des données à l'utilisation des connaissances, "l'informatique dans les formations du secteur tertiaire,
Actes du Colloque franco-néerlandais de Royaumont,MEN/DCRI,
Mai 1986

[GRAN,88] GRANDBASTIEN M.
Towards a better use of computers in computer science teaching
ECCE 88, Lausanne, Computers in Education, North Holland, 1988

[GRAN,86] GRANDBASTIEN M., MAROLDT J.
Towards an Expert System for Troubleshooting Diagnosis in large industrial plants , 1st.Int.Conf.on applications of AI in engineering problems,
Southampton, 1986

[GRAN,87] GRANDBASTIEN M., MAROLDT J.
Conception d'un système d'aide au diagnostic pour les installations industrielles, réalisation d'un prototype dans Systèmes experts et maintenance, GABRIEL M.& RAULT J.C. eds., Masson , 1987

[GREE,77] GREEN C.
A summary of the PSI program synthesis system
5ème IJCAI,pp.380-381,1977

[GREE,78a] GREEN C.
The design of the PSI program synthesis system
3rd.Software Engineering Conference,pp.4-18,1978

[GREE,78b] GREEN C.
On program synthesis knowledge
Artificial Intelligence,vol.10,n°3,pp.241-279,1978

[GRES,82] GRESSE C., GUIHO G
Proposition d'un environnement interactif pour faire de la construction assistée de programmes Colloque AFCET-Génie Logiciel, pp.65-75, 1982

[GRES,83] GRESSE C.
Automatic programming from data types decompositions patterns
8th. IJCAI, Karlsruhe, pp.37-39, 1983

[GRES,84] GRESSE C.
Contribution à la programmation automatique - Caty : un système de construction assistée de programmes, Thèse d'Etat, Orsay, 1984.

[GRI,81] GRIES D.
The Science of Programming
Springer Verlag, 1981

[GUT,77] GUTTAG J.
Abstract data types and the development of data structures
CACM, 20, vol.6, pp.396-404, 1977

[GUT,80] GUTTAG J.
Notes on type abstraction Proceedings of Specifications of Reliable Software IEEE TSE,vol.6,n°1,pp.13-23,1980

[GUYA,84] GUYARD J., JACQUOT J.P.
MAIDAY: An Environment for Guided Programming with a Definitional Language 7th. Conference on Software Engineering, Orlando,USA, 1984

[GUYO,86] GUYOT J., VIAL C.
Arbres, tables et algorithmes.
Eyrolles, Paris, 1986.

[HAR,73] HARTLEY J.R., SLEEMAN D.H.
Toward more intelligent teaching systems. Int.J.of Man Machine Studies, vol.5, pp.215-236, 1973

[HAT,88] HATON J.P., HATON M.C. , LUPIN B., VION E. , RIGGI P.
Formation aux consignes d'exploitation en métallurgie : le système CONSOL, les SE et leur application, 8èmes journées internat. Avignon, 1988

[HAY,83] HAYES-ROTH F., WATERMAN D.A., LENAT D.B. (ed.)
Building Expert Systems.
Addison Wesley Pub. Cie, Menlo Park, 1983

[HAY,86] HAYASHI S.
PX, a system for extracting programs from proofs
IFIP Conference on formal description of programming concepts, 1986

[HEI,76] HEIDORN J.
Automatic programming through natural language dialogue: a survey, IBM Journal of research and development, vol.20,n°4,pp.302-313,1976

[HER,85] HERMANT C.
Enseigner, Apprendre avec l'Ordinateur.
Cedic / Nathan, Paris , 1985

[HIB,81] HIBBARD P.,HISGEN A.,ROSENBERG J.,SHAW M.,SHERMAN M.
"Studies in ADA style" Springer Verlag, 1981

[HIL,87] HILL W.L.
Machine learning for software reuse
IJCAI, 1987

[HOA,78] HOARE C.A.R.
Data structures, in Current Trends in Programming Methodology
YEH (ed.),Prentice Hall, 1978

[HOC,78] HOC J.M.
La programmation informatique comme situation de résolution de problèmes
Thèse 3ème cycle, univ. René Descartes, Paris, 1978

[HOR,76] HOROWITZ E., SAHNI S.
Funtamentals of Data Structures
Pitman, 1976

[HSI,84] HSIANG J., SRIVAS M.
A PROLOG environment for developing and reasoning about data types
Dep. of computer science, Stony Brook, 1984 internal report,1984

[IASI,87] Equipe Intelligence artificielle et Systèmes d'Inférences du LRI
MORSE, Manuel d'utilisation, Nov. 1987

[ICH,85] ICHBIAH J
ADA: l'heure des applications
TSI,pp.185-188,1985

[ICH,84] ICHBIAH J.
Ada: Past, Present, Future
CACM,27, vol.10, pp., 1984

[JACK,75] JACKSON M.A.
Principles of Program Design, Academic Press, London, 1975

[JACQ,78] JACQUET Paul
La généricité comme outil d'abstraction dans les langages de programmation
Congrès AFCET-TTI, Gif sur Yvette, 1978

[JOH,85] JOHNSON W.L., SOLOWAY E.
PROUST: An Automatic Debugging for Pascal Programs. BYTE, vol.10, pp.179-190,1985

[JOH,85] JOHNSON W.L., SOLOWAY E.
PROUST: Knowledge-based program understanding, IEEE Transactions on Software Engineering, vol.SE 11,n°3,pp.267-275, 1985

[JOH,84] JOHNSON W.L., SOLOWAY E.
PROUST: Knowledge-Based Program Understanding. ,7th.international conference on Software Engineering, pp.369-380, 1984

[JOR,77] JORRAND P.
Very high Level Languages : some aspects of the evolution of language design, International Computing Symposium, North Holland, 1977

[KAN,77] KANT E.
The selection of efficient implementations for a high level language
Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, pp.140-146, Aug.1977

[KAN,83] KANT E.
On the efficient synthesis of efficient programs.
AI magazine, pp.283-305, 1983

[KAN,81] KANT E.,BARSTOW D.R.
The refinment paradigm: the interaction of coding and efficiency knowledge in program synthesis, IEEE TSE,vol.7, n°5,pp.458-471,1981

[KAS,86] KASSEL G.
Expliquer, c'est raisonner sur le raisonnement : le système CQFE
6ème journées internat. Systèmes Experts et Applications, Avignon 1986, pp.973-990, 1986

[KIR,85] KIRSCH P. et équipe "Systèmes Experts" du LRI
MORSE: manuel de référence
Rapport LRI n°255, Déc.1985

[KNU,73a] KNUTH D.E.
The Art of Computer Programming: Fundamentals Algorithms
Addison Wesley, 1973

[KNU,73b] KNUTH D.E.
The Art of Computer Programming: Sorting and Searching
Addison Wesley,1973

[KOF,75] KOFFMAN E.B.
Artificial Intelligence and automatic programming in CAI
Artificial Intelligence, vol.6, pp.215-234, 1975

[KRI,87] KRIVINE J.L.,PARIGOT M.
Programming with proofs
6th.Symposium on Computation Theory, Wendish-Rietz, November 1987

[LAUB,81] LAUBSCH J.,EISENSTADT M.
Domain specific debugging aids for novice programmers,
7eme IJCAI, pp.964-969, 1981

[LAURE,88] LAURENT JP et al.
Schéma pour la description et l'évaluation de SE et d'outils pour développer
Actes des Journées PRC. Intelligence artificielle, Teknea,Toulouse, 1988

[LAURE,87] LAURENT JP;THOME F; AYELJ; ZIEBELIN D;
Kee, Knowledge Craft et Art:Eval comparative de 3 outils de développement de systèmes experts, Revue d'IA , Vol.2, HERMES, 1987

[LAURI,78] LAURIERE JL A language and a problem for stating and solving Combinatorial problems
Artificial Intelligence n°10, pp. 29 -127, 1978

[LAURI,86a] LAURIERE J.L.
Un langage déclaratif: SNARK
TSI n°3,pp.141-172,1986

[LAURI,86b] LAURIERE J.L.
Intelligence Artificielle: résolution de problèmes par l'homme et par la machine, Eyrolles, Paris, 1986

[LAW,86] LAWLER R.W., YASDANI M.
Artificial Intelligence and Education vol.1, Learning Environments and Tutoring Systems, 1986

[LE VER,82] LE VERRAND D.
Le langage ADA: Manuel d'évaluation , Dunod, Paris, 1982

[LEC,85] LECARME O., ROUSSEAU R.
Langages de programmation; types et modularité
Collection Mémoire Vive, Agence de l'Informatique, Paris, 1985

[LEF,84] LEFEVRE J.M.
EAO : Guide pratique de l'enseignement assisté par ordinateur
Cédic / Nathan, Paris, 1984

[LEV,84] LEVY N.
Outils d'aide à la construction et transformation de types abstraits algébriques
Thèse 3° cycle, Université Nancy 1, 1974

[LEV,86] LEVY N., SOUQUIERES J. Spécification des types et opérations de base dans le système SACSO, Document interne, CRIN, 1986

[LEV,87] LEVY N., PIGAGNIOL A., SOUQUIERES J.,
Specifying with SACSO 4 th. intern. workshop on Software Specification and Design Monterey, Californie, USA, 1987

[LIS,80] LISKOV B.
Programming with abstract data types in Programming Language Design
IEEE Computer Society Press, Los Alamitos, Calif.,1980

[LIS,74] LISKOV B., ZILLES S.
Programming with abstract data types
ACM SIGPLAN Notices, vol.9, n°4, pp.50-59, 1974

[LIS,86] LISKOV B.,GUTTAG J.
Abstraction and specification in program development
Cambridge, MIT Press, 1986

[LIS,77] LISKOV B.H., SNYDER A., ATKINSON R. SCHAFFERT C.
Abstraction mechanisms in CLU, CACM, 20, vol.8, pp. 564-576, 1977

[LIV,78] LIVERCY C.
Théorie des Programmes, Dunod, Paris, 1978

[LUC,83a] LUCAS M.
Algorithmique et Représentation des Données, tome 2, Evaluation, arbres, graphes, analyses de textes, Masson, Paris, 1983, 2eme édition, 1986

[LUC,83b] LUCAS M., PEYRIN J.P., SCHOLL P.C.
Algorithmique et représentation des données, tome 1, Files, automates d'états finis, Masson, Paris, 1983

[Mc NEI,86] Mac NEILE AT.
Jackson syst devel.(JSD)in information systems design methods:Improve the practise, Elsevier Sc.Pub, North Holland, IFIP, 1986

[MAD,87] MADAULE F. et al.
Systèmes d'enseignement assisté par ordinateur, étude comparative
TSI, vol.6, n°1, 1987

[MAN,75] MANNA Z., WALDINGER R.
Knowledge and reasoning in program synthesis
Artificial Intelligence, vol.6, pp.175-208, 1975

[MAN,79] MANNA Z., WALDINGER R.
Synthesis: dreams-programs
IEEE TSE, vol.5, n°4, pp.294-327, 1979

[MAN,80] MANNA Z., WALDINGER R.
A deductive approach to program synthesis
ACM Toplas, vol.2, n°1, pp.90-121, 1980.

[MAN,71] MANNA Z., WALDINGER R.
Towards automatic program synthesis
CACM, vol.14, n°3, 1971

[MAR,86] MARTIN J.J.
Data types and data structures
Prentice Hall, (UK),1986

[MAT,85] MATHIEU J., THOMAS R
Manuel de psychologie, Vigot, 1985

[MAY,82] MAYOH B.
Problem Solving with ADA
John Wiley & Sons, 1982.

[MEY,85a] MEYER B.
ADA: le langage et l'environnement, 1975-1985, interview de D.FISCHER (Gensoft) et J.BUXTON (King's College), TSI, vol.4,n°2, 1985

[MEY,85b] MEYER B.
Etapes sur le chemin du génie logiciel
Thèse d'Etat, Université Nancy 1, 1985

[MEY,87a] MEYER B.
Genericity, inheritance and type checking
TR-E1-8/GI, Interactive Software Ingineering Inc., 1987

[MEY,87b] MEYER B.
EIFFEL : Programming for Reusability and Extendibility
ACM Sigplan Notices, March 1987

[MEY,78] MEYER B., BAUDOIN C.
Méthodes de Programmation
Eyrolles, 1978

[MIL,82] MILLER J.H., STONER T.T.
Trends and prospects for micro-computer based education., Int.Journal of Man-Machine Studies, vol.17, pp.143-148, 1982

[MIL,82] MILLER M.L.
A Structured Planning and Debugging Environment for Elementary Programming, in Intelligent Tutoring Systems, pp.119-133. SLEEMAN D., BROWN J.S. (eds.), Acad. Press, London, 1982

[MOR,88] MORINET-LAMBERT J.
Notice du logiciel SAIDA, rapport CRIN 88-R-071, 1988

[MYC,84] MYCROFT A., O'KEEFE R.A. A polymorphic type system for Prolog
Artificial intelligence, vol.23, pp.295-307, 1984

[NEI,84] NEIGHBORS J.M.
The Draco Approach to Constructing software from reusable components, IEEE TSE,vol.10,n°5,pp.564-574, 1984

[NEW,63] NEWELL A.,SIMON H.A.
GPS, A Program that simulates Human Thought., in Computers and Thought, Mac Graw Hill, New York, 1963.

[NIC,88] NICAUD J.F., VIVET M. Les tuteurs intelligents, réalisations et tendances de recherches. TSI, vol.7, n° 1, Dunod, Paris, 1988

[O'SHEA,83] O'SHEA T., SELF J.
Learning and Teaching with the Computer, Artificial Intelligence in Education. Harvester Press, Brighton, UK, 1983

[PAI,71] PAIR C.
Structures d'informations
Ecole d'Informatique de l'AFCET, 1971

[PAI,74] PAIR C.
Structures de Données et Algorithmes Fondamentaux. Cours de 2° année (polycopié) ENSMIN, Nancy,1974

[PAI,79] PAIR C.
La construction des Programmes, RAIRO Informatique, vol 13 -2, 1979

[PAI,86] PAIR C.
Programmation et structures de données
Polycopié, Ecole des Mines de Nancy, 1986

[PAI,88] PAIR C., MOHR R., SCHOTT R.
Construire les algorithmes, les améliorer, les connaître, les évaluer
Dunod, 1988

[PAL,88] PALIES O.
Métaconnaissance pour la modélisation de l'élève. Contribution au diagnostic par système expert, thèse, Paris VI , 1988

[PAP,80] PAPERT S.
Jaillissement de l'esprit , Flammarion, 1980

[PAR,86] PARTSCH H., PEPPER P.
Program transformations expressed by algebraic type manipulation
TSI, vol.5, n°3, pp.197-212, 1986

[PAS,87] PASSARDIERE B. de la, MADAULE F.
Apports de l'EAO dans l'enseignement de l'informatique. Congrès francophone EAO 87, Cap d'Agde, pp.235-246, 1987

[PIT,85] PITRAT J.
MACISTE ou comment utiliser un ordinateur sans utiliser de programme?
colloque intelligence artificielle,Toulouse, pp.223-240,GR22 CNRS, 1985

[PUN,84] PUN L.
Systèmes industriels d'intelligence artificielle: outils de productique.
Editions Tests, Paris, 1984

[QUE,80] QUERE M.
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 d'Etat, Université Nancy 1,1980

[QUE,85] QUERE M.
Expert systems: Towards CAI of the future
WCCE 85, Norfolk, USA, 1985

[QUE,83] QUERE M.
Outils pour l'enseignement assisté par ordinateur, Bulletin de l'association EPI, 1983

[RAU,83] RAULT J.C.
Les systèmes experts: perspectives industrielles
Informatique et Gestion, Septembre 1983

[REG,85a] REGOURD J.P.
LOGO-84 : A teaching language Actes du Colloque Européen LOGO
E.N.S. de St. Cloud, Sept.1985

[REG,85b] REGOURD J.P.
Apprentissage autonome et Intelligence Artificielle Thèse 3° cycle, Université Paris 6 et LITP, Décembre 1985

[REG,87] REGOURD J.P.
GAMETE : Le manuel de référence, Rapport LITP n° 87-9, 1987

[REG,88] REGOURD J.P.
Gamète
TSI, vol.7,n°1, 1988

[REIS,85] REISER A.J., ANDERSON J.R., FARREL G.
Dynamic student modelling in an intelligent tutor for lisp programming
IJCAI 85, 1985

[REIT,84] REITMAN W. (ed.)
Artificial Intelligence Applications for Business
Norwood, N.J.,Ablex Pub. Corporation, 1984.

[REM,80] REMY J.L.
Construction, évaluation et amélioration systématique des structures de données, RAIRO informatique théorique, vol.1,1980

[RIC,78] RICH C., SHROBE H.
Initial report on a LISP programmer's apprentice
IEEE TSE, vol. 4 , Nov 78, pp. 456 -466, 1978

[RIC,81a] RICH C.
Inspection Methods in Programming
Ph.D. Thesis, MIT AI lab., juin 1981

[RIC,81b] RICH C.
A formal representation for plans in the programmer's apprentice
Proceedings IJCAI 7, pp.1044-1052 , 1981

[RIC,85] RICH C.
The layered architecture of a system for reasoning about programs
Proceedings IJCAI 9, pp.540-546, 1985

[RIC,78] RICH C., SHROBE H.
Initial report on a LISP programmer's apprentice. IEEE Transactions on Software Engineering, vol.4, pp.456-466, 1978

[RIC,86] RICH C., WATERS R.C. (eds.)
Readings in Artificial Intelligence and Software Engineering,
Morgan Kaufmann Pub.,1986

[ROC,85] ROCKMORE A.J. Knowledge-based software turns specifications into efficient programs, Electronic Design, 33, 17, pp.105-110, July 1985

[ROS,86] ROSEN J.P.
Un package de gestion de menu : expérience de développement de composant logiciel avec quelques morales, Bulletin BIGRE+GLOBULE

[RUT,76] RUTH G.
Intelligent program analysis
Artificial Intelligence, n°7,pp.65-85,1976

[RUT,78] RUTH G.
Protosystem I: An automatic programming sysyem prototype
AFIPS Conf. proc.,pp.675-681,1978

[SAF,85] SAFAR B.
Explications dans les systèmes experts
5° Journées Internationales les systèmes experts et leurs applications
Avignon, 1985

[SCHE,83] SCHERLIS W.L., SCOTT D.S.
First steps through inferential programming
IFIP congress, North Holland, Amsterdam, 1983

[SCHL,84] SCHLIENGER F.
Un environnement de Programme ADA intègrant des Spécifications Algébriques. Thèse de 3° cycle, Orsay , 1984.

[SCHOL,79] SCHOLL P.C.
Vers une programmation systématique: étude de quelques méthodes, techniques et outils, Thèse d'Etat, Université de Grenoble, 1979

[SCHOL,83] SCHOLL P.C.
Algorithmique et représentation des données, tome 3, Récursivité et arbres
Masson, 1983

[SCHOM,82] SCHOMAN K., ROSS D.T.
Structured analysis for requirement definition
IEEE TSE SE-3(1), pp.6-15, 1982

[SCHON,86] SCHONBERG E., SCHIELDS D.
From prototype to efficient implementation: a case study using SETL and C, Proc. 19° Hawai Intern. Conf. on System Science, BRUCE SHRIVER (eds), IEEE, 1986

[SCHON,81] SCHONBERG E.,SCHWARZ J.T.,SHARIR M.
An automatic technique for selection of data representations in SETL programs, ACM Toplas,3(2),pp.126-143,1981

[SCHW,74] SCHWARTZ J.T.
Automatic and semi automatic optimization on SETL
ACM SIGPLAN Notices, vol.9, n°4, pp.43-49, 1974

[SCHW,86] SCHWARZ J.T., DEWAR R.B.K., DUBINSKY E., SCHONBERG E.
Programming with sets. An introduction to SETL
Springer Verlag , 1986

[SCHW,79] SCHWARZ J.T., SHARIR M.
Experience with automatic data structure selection in the SETL system presented at the 2nd Program Transformation Workshop HARVARD University, Mass., Sept.1979

[SCW,87] SCWABE D.,MARTINS R.C.B.,PESSOA T.E.C.
An intelligent tool for program development: an expert assistant in Jackson's JCP. Journées Systèmes experts, Avignon, 1987

[SLE,82] SLEEMAN D., BROWN J.S. (eds.)
Intelligent tutoring systems
Academic Press, London, 1982

[SLE,87] SLEEMAN D.
Some challenges for intelligent tutoring systems
IJCAI 1987,pp.1066-1068, 1987

[SMO,83] SMOLIAR S.W.
Software Specifications Data Bases and Knowledge Bases
IFIP, North Holland,pp.219-221,1983

[SOL,81] SOLOWAY E.M.,WOOLF B.,RUBIN E.,BARTH P.
MENO II ; An intelligent tutoring system for novice programmers
7 th. IJCAI, Vancouver, pp.975-977, 1981

[SOL,82] SOLOWAY E., EHRLICH K.
"Tacit programming Knowledge"
4th. cognitive science conference, Ann Arbor, 1982

[SOL,83a] SOLOWAY E., EHRLICH K.
What DO programmers reuse? Theory and experiment Proceedings of the Workshop on Reusability in Programming ITT Programming, Stratford, Conn.,1983

[SOL,83b] SOLOWAY E.M.,RUBIN E.,WOOLF B.,BONAR J., JOHNSON W.L.
MENO II: An intelligent based programming tutor
J.Computer based instruction, vol.10, pp.20-34, 1983

[SOL,84] SOLOWAY E.
Empirical Studies of programming languages
IEEE TSE vol.10,n°5,pp.595-609,1984

[SOW,84] SOWA J.F.
Conceptual Structures,Information Processing in Mind and Machine.
Addison Wesley,1984.

[SRI,82] SRIVAS M.K.
Automatic Synthesis of Implementation for Abstract Data Types from Algebraic Specifications MIT/LCS/Tech.Rep. n°276, Lab. for Computer Science, June 1982

[STE,82] STEFIK M. et al.
The Organization of Expert Systems, A Tutorial.
Artificial Intelligence, vol.18, n°2, March 1982

[SWA,82] SWARTOUT W., BALZER R.
On the inevitable intertwining of specification and implementation.
CACM, vol.25, n°7, pp.438-440, 1982

[TAR,83] TARDIEU H., ROCHFELD A.,COLLETI R.
La Méthode MERISE : Principes et Outils, Editions d'organisation, 1983

[VER,86] VERILOG
LOGISCOPE, manuel d'utilisation, Société Verilog, Toulouse, 1986

[VIV,84] VIVET M.
Expertise mathématique et informatique - CAMELIA: un logiciel pour raisonner et calculer, thèse d'Etat, Paris VI, Juin 1984

[VIV,87a] VIVET M.
Hierarchy of knowledges in an intelligent tutoring system, how to take account of the student ? European seminar on intelligent tutoring systems, Tubingen, Oct.1987

[VIV,87b] VIVET M.
Systèmes experts pour enseigner: métaconnaissances et explications
Colloque Mari-Cognitiva, 1987

[WAL,69] WALDINGER R.J., LEE R.C.
"PROW": A step toward automatic program writing, Proceedings 1st IJCAI, pp.241-252, 1969

[WAN,80] WAND M.
Continuation-based program transformation strategies
JACM, vol.27, n°1, pp.164-180, 1980

[WAR,75] WARNIER J.D.
Les procédures de traitement et leurs données (LCP)
Editions d'Organisation, Paris, 1975

[WAR,72] WARNIER J.D., FLANAGAN B.
Entraînement à la programmation, tomes 1 et 2
Editions d'organisation, Paris, 1972

[WATERM,86] WATERMAN D.A.
A Guide to Expert Systems.
Addison Wesley Pub. Cie, 1986

[WATERS,82] WATERS R.C.
The Programmer's Apprentice: Knowledge Based Program Editing,
IEEE Trans. on Software Engineering, 8, 1, 1982

[WATERS,86] WATERS R.C.
KBEmacs : Where's the AI? AI Magazine,vol.7, n°1, pp.47-56, 1986

[WATERS,86] WATERS R.C.
The programmer's apprentice: A session with KBEmacs. IEEE Transactions on Software Engineering, vol.11, pp.1296-1320, 1986

[WER,84] WERZ H.
Etude,Réalisation et Evaluation d'un environnement de Programmation
Université de Paris VIII - Vincennes, Thèse d'Etat, 1984

[WER,85] WERTZ H.
Intelligence artificielle, application à l'analyse de programmes
Masson, 1985

[WHI,83] WHITE J.
On the multiple implementation of abstract data types within a computation
IEEE TSE vol.9,n°4, pp.395-410, 1983

[WIN,84] WINSTON P.H.
Artificial Intelligence.
Addison Wesley Pub. Cie, Menlo Park, 1984.

[WIR,76] WIRTH N.
Algorithms + Data Structures = Programs
Prentice Hall, 1976

[WIR,71] WIRTH N.
Program development by stepwise refinement.
CACM, vol.14, pp.221-227, 1971

[WIR,77] WIRTH N.
Introduction à la programmation systématique
Masson, Paris, 1977

[WIR,84] WIRTH N.
Programmer en Modula 2,
Presses polytechniques Romandes, 1984

[WIR,86] WIRTH N.
Algorithmes et Structures de données
Eyrolles, Prentice hall 86, (mise à jour de l'édition de 76), 1987

[WUL,75] WULF W.A, LONDON R.L, SHAW M.
Abstraction and Verification in ALPHARD, in New Directions in algorithmic languages, S.A Shuman ed., 1975

[YAS,86a] YASDANI M.
Intelligent tutoring systems survey.
AI Review, vol.1, pp.43-52, 1986

[YAS,86b] YASDANI M.
Intelligent tutoring systems : an overview
Expert Systems, vol.3,n°3,pp.154-163,1986

[YEH,78] YEH R. (ed.)
Current trends in programming methodology
Englewood Cliffs,N.J., Prentice Hall, 1978

Références relatives à l'avant-propos

[1] GRANDBASTIEN M.
Un Programme qui résout formellement des équations trigonométriques par des méthodes heuristiques. Thèse de 3° cycle, Université Paris VI, 1974

[2] GRANDBASTIEN M.
Résolution automatique de problèmes et enseignement assisté par ordinateur:une application en trigonométrie, 8ème Congrès Internat. de Cybern. de Namur, 1976

[8] GRANDBASTIEN M. Méthodes de construction de systèmes à bases de connaissances, Actes des Journées du PRC-GRECO Intelligence Artificielle, Aix-les-Bains, Cepadues Editions, Toulouse, 1986

[3] GRANDBASTIEN M.
Le développement de l'intelligence artificielle concerne tous les enseignants
Revue de l'association EPI, numéro spécial "Systèmes experts", déc.1984

[4] GRANDBASTIEN M.
Evolution de l'informatique: du traitement des données à l'utilisation des connaissances, "l'informatique dans les formations du secteur tertiaire",
Actes du Colloque franco-néerlandais de Royaumont,MEN/DCRI, mai 1986

[5] GRANDBASTIEN M.
Les applications de l'intelligence artificielle
Sessions de formation continue d'ingénieurs, ENSMIN et DPIC,1986

[6] GRANDBASTIEN M., MAROLDT J.
Towards an Expert System for Troubleshooting Diagnosis in large industrial plants
1st.Int.Conf.on applications of AI in engineering problems,Southampton, 1986

[7] GRANDBASTIEN M.,MAROLDT J.
Conception d'un système d'aide au diagnostic pour les installations industrielles, réalisation d'un prototype dans Systèmes experts et maintenance,
GABRIEL M.& RAULT J.C., Masson , 1987

[9] LIVERCY C. (nom collectif)
Théorie des programmes, Dunod, 1978

[10] PAIR C.
La construction des programmes, RAIRO Informatique, vol.13-2, 1979

[11] DUCRIN A.
Programmation, tomes 1 et 2, Dunod, 1984

[12] IREM
Algorithmique, Informatique, Mathématiques en 2° cycle, IREM de Lorraine,1984

[13] Collection "Fenêtre Active"
(consacrée aux logiciels pédagogiques et à leurs applications dans la classe)
Publications du CRDP de Nancy, 99 rue de Metz, 54000 - Nancy

[14] GRANDBASTIEN M. et al.
Rapports du Comité Scientifique National chargé du suivi de l'option informatique des lycées au Directeur des lycées, Ministère de l'Education Nationale,1984, 1985, 1987

NOM DE L'ETUDIANT : Mme GRANDBASTIEN Monique

NATURE DE LA THESE : DOCTORAT D'ETAT ES SCIENCES MATHEMATIQUES

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le - 5 JUIL. 1988 n° 1189

LE PRESIDENT DE L'UNIVERSITE DE NANCY I

RESUME

Le niveau croissant de complexité des applications à développer en informatique d'une part, l'évolution des méthodes, des logiciels et des matériels d'autre part, modifient l'activité de construction de programmes. Dans ce contexte en évolution rapide, les responsables de formation s'interrogent : que faut-il enseigner aux futurs professionnels de l'informatique dans le domaine de la construction des programmes ? Comment faut-il enseigner ? Peut-on proposer des outils qui facilitent la tâche des élèves et des enseignants ?

Ce travail est une contribution à la réflexion commune sur l'étude de la programmation et de son enseignement. Il aborde plus précisément le thème de l'écriture d'algorithmes utilisant des types abstraits et des opérations définies sur ces types et le thème du choix de représentations des types permettant d'implanter efficacement les opérations. Il propose avec les bases de connaissances une approche originale pour l'enseignement de techniques jusqu'ici assez peu systématisées.

Dans la première partie, les problèmes soulevés par la construction de programmes par niveaux d'abstractions successifs sont rappelés à l'aide d'exemples ; ils sont ensuite placés dans le contexte des travaux actuels sur la programmation et les tentatives d'automatisation de cette activité. Le rôle d'assistance que l'ordinateur peut jouer dans l'enseignement, et notamment dans l'enseignement de l'informatique, est également décrit.

La seconde partie de l'ouvrage est consacrée à l'étude de la matière à enseigner, les types abstraits à retenir et les représentations usuelles, le mode d'expression des algorithmes et des programmes. Le langage ADA qui permet à la fois d'écrire des procédures sur des objets abstraits et des programmes efficaces implantant ces procédures a été choisi comme support de l'application développée.

La troisième partie porte sur la conception d'un système d'enseignement sur les types abstraits et leurs représentations. Ce système repose sur trois composants principaux : le premier est une bibliothèque de types et de représentations pour ces types ; le second est une base de connaissances formalisant les critères de choix de représentations en fonction des caractéristiques de l'environnement, de l'objet à représenter et des opérations effectuées dans l'algorithme ; le troisième propose différents modes d'utilisation de cette base de connaissances traduisant des préoccupations d'enseignement différentes et exprimant une part d'expertise pédagogique.

Le logiciel SAIDA réalisé à partir de cette étude est décrit dans la quatrième partie. Il est construit sous UNIX sur machine SUN de Matra Datasystème à partir des logiciels Suntools, MORSE et MENTOR-ADA ; il utilise le compilateur et les bibliothèques VERDIX/ADA. C'est un environnement de travail ouvert destiné à des enseignants et à des élèves. Aux enseignants, il donne la possibilité de paramétrer l'environnement de travail en fonction de la progression pédagogique qu'ils ont choisie, des modes de raisonnements qu'ils veulent privilégier et d'autres caractéristiques de leur population d'élèves. Les élèves y construisent des programmes en utilisant la bibliothèque de types et bénéficient de l'assistance du système à base de connaissances pour apprendre à choisir des implantations efficaces des objets abstraits de leurs algorithmes.

La conception et la réalisation de cette application ont conduit à beaucoup de questions nouvelles, tant sur le plan de la formalisation des critères de choix de représentation que sur celui de l'utilisation de bases de connaissances formalisant l'expertise d'un domaine dans l'enseignement. Une synthèse de ces questions et des voies de recherche qu'elles peuvent ouvrir est exposée en conclusion.

MOTS - CLES : système à base de connaissances, programmation, enseignement de la programmation, types de données, représentation physique de données.